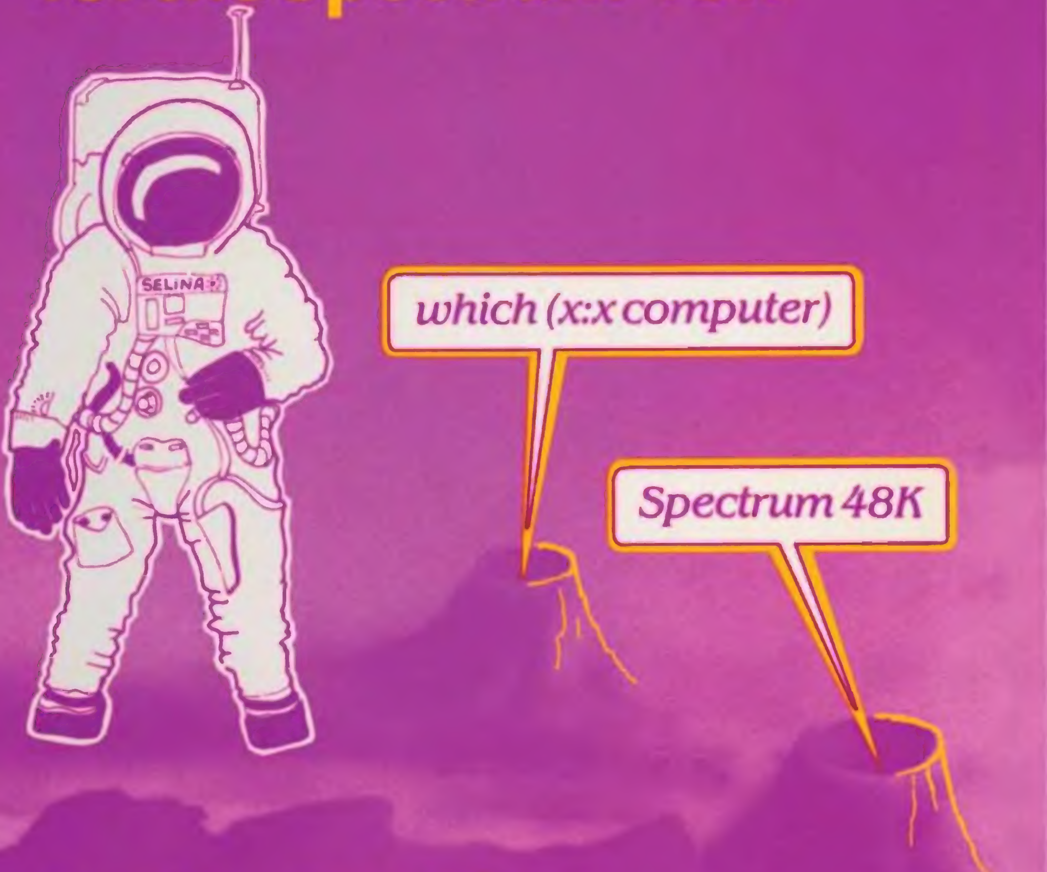
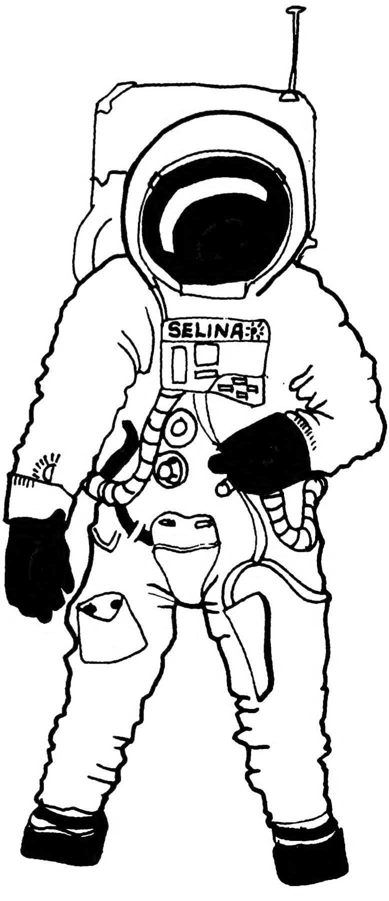
THE PURPLE PLANET

Micro-PROLOG for the Spectrum 48K



Serafim Gascoigne


SELINA

The Purple Planet
Micro-PROLOG for the Spectrum 48K

Macmillan Microcomputer Books

General Editor: Ian Birnbaum
(General Adviser (Microelectronics in Education)
Education Department, Humberside County Council)

Advanced Graphics with the Acorn Electron
Ian O. Angell and Brian J. Jones
Advanced Graphics with the BBC Model B Microcomputer
Ian O. Angell and Brian J. Jones
Interfacing the BBC Microcomputer
Brian Bannister and Michael Whitehead
Assembly Language Programming for the Acorn Electron
Ian Birnbaum
Assembly Language Programming for the BBC Microcomputer (second edition)
Ian Birnbaum
Using Your Home Computer (Practical Projects for the Micro Owner)
Garth W. P. Davies
Microchild – Learning through LOGO
Serafim Gascoigne
A Science Teacher's Companion to the BBC Microcomputer
Philip Hawthorne
Beginning BASIC with the ZX Spectrum
Judith Miller
Using Sound and Speech on the BBC Microcomputer
Martin Phillips
File Handling on the BBC Microcomputer
Brian J. Townsend
Good BASIC Programming on the BBC Microcomputer
Margaret White

Other Books of related interest

Advanced Graphics with the IBM Personal Computer
Ian O. Angell
Advanced Graphics with the Sinclair ZX Spectrum
Ian O. Angell and Brian J. Jones
Beginning BASIC
Peter Gosling
Continuing BASIC
Peter Gosling
Practical BASIC Programming
Peter Gosling
Program Your Microcomputer in BASIC
Peter Gosling
More Real Applications for the ZX81 and ZX Spectrum
Randle Hurley
The Alien, Numbereater, and Other Programs for Personal Computers
John Race
Computer Literacy: A beginners' guide
Vincent Walsh

Mastering Computers
G. G. L. Wright
Mastering Computer Programming
P. E. Gosling
Mastering Data Processing
J. Bingham
Mastering COBOL
R. Hutty
Mastering Pascal Programming
E. Huggins

The Purple Planet

Micro-PROLOG for the Spectrum 48K

Serafim Gascoigne

MACMILLAN

First published 1985

Published by
Higher and Further Education Division
MACMILLAN PUBLISHERS LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

British Library Cataloguing in Publication Data
Gascoigne, Serafim
The Purple Planet: Micro-PROLOG for the Spectrum 48K.
1. Sinclair ZX Spectrum (Computer) — Programming
2. Micro-PROLOG (Computer program language)
I. Title
001.64'24 QA76.8.S625

ISBN 978-1-349-07862-2 ISBN 978-1-349-07860-8 (eBook)
DOI 10.1007/978-1-349-07860-8

Contents

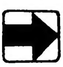
Preface

Would you like to use your Spectrum for more than games and other types of ready-made software? Are you interested in writing your own computer programs? Programming is not as difficult as some people might believe. What is more, some highly sophisticated computer languages are available for the home micro. Micro-PROLOG is one of them. It comes from a new science called Artificial Intelligence. This is the science of building ´intelligent´ machines.

Through Micro-PROLOG you too can explore this fascinating world of ´intelligent´ computers. You can write your own expert system and ask your computer questions. It can ask you questions and give you advice as well!

You can create your own adventure games, a fantasy world with its own rules or invent a fault-finding system to test electrical circuits. Whatever ´expert´ knowledge you have, from collecting things to cooking continental dishes, Micro-PROLOG provides you with an ´intelligent´ database language, through which you and your computer can ask each other questions. It also allows you to build your own language at the keyboard. Whether you are a complete beginner to computing or are considering a career in computer technology, Micro-PROLOG is a must. It is one of the languages of the future. It has for instance been chosen by the Japanese government to be the main language of their fifth generation computers. The only limitation to Micro-PROLOG is your own imagination!

To make it easier for the reader to identify different sections of program listings shown in the text, the following symbols are used to indicate the different parts of the program:

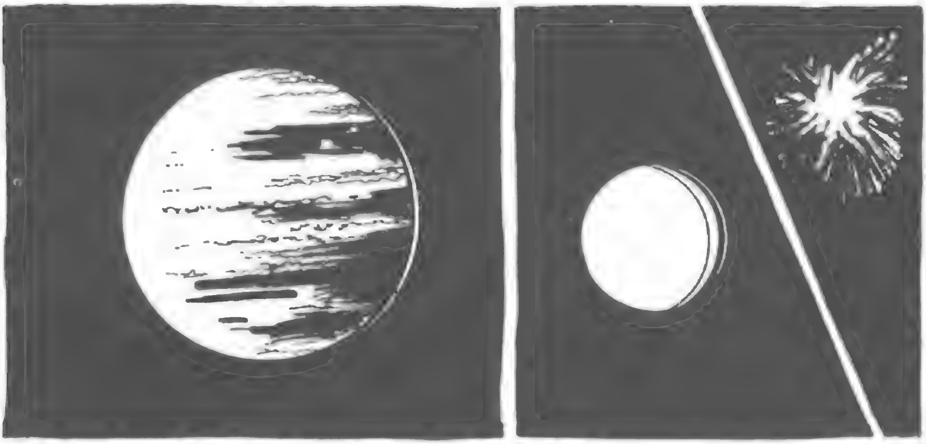


For new items of data

For any rules of syntax

For any interrogation routines

1 The Purple Planet



Loading Micro-PROLOG

Before you begin programming in Micro-PROLOG you will need to load the cassette as follows.

1. Place the Micro-PROLOG cassette in your cassette player and press the LOAD key on the SPECTRUM. Next type "PROLOG" and press the ENTER key and the play button on your cassette player.
After a minute, the following copyright message will be displayed

```
Spectrum micro-PROLOG T1.0
      C 1983 LPA Ltd.
           XXXX
            &.L
```

2. Now type LOAD SIMPLE. SIMPLE is a modified version of Micro-PROLOG, which is used throughout this book. It is nearer to ordinary English than Micro-PROLOG and is therefore easier to understand. Once SIMPLE has been loaded, the prompt sign &. will appear. You are now ready to begin.

Describing your program

Most computer languages consist of instructions or commands to the computer such as PRINT, DRAW, PLOT, or MAKE. They tell the computer to do this or that. In order to write a conventional computer program you are normally required to know how the program works. Although this may seem obvious to you, in Micro-PROLOG you do not have to know how the program works, but rather what it does.

In Micro-PROLOG, or simply PROLOG as I shall refer to the language throughout this book, there are just a few commands. For the most part PROLOG works with descriptions rather than instructions. You do not, for example, write a list of instructions as you would if you were using BASIC. The information or data you have collected is put into your program in the form of descriptions, that is sentences, that describe what your program does. For example, a typical line of a program in PROLOG could be

```
Zorg likes Selina
```

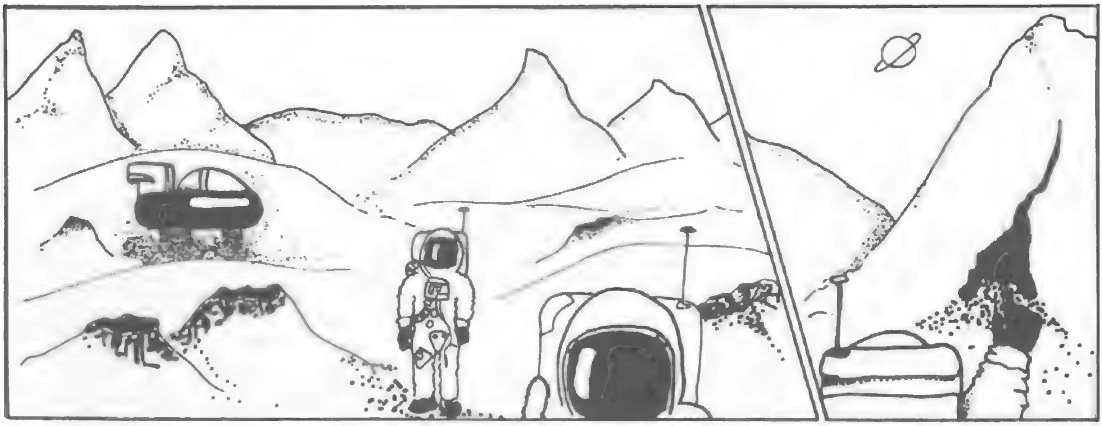
This does not tell the computer to do anything. It simply states a fact. Another typical line could be

```
Owls belong-to Land-birds
```

This program line is a description about owls, not an instruction to the computer to do something.

But surely the computer must do something with the data (information) that has been fed into it! It does. It stores this data in its memory. You can then ask questions about this data.

PROLOG, which stands for PROGramming in LOGic, is what is known as a database language. Your descriptions or data can be stored in the form of descriptive statements, which can then be manipulated according to specific ´rules´ which you have invented. PROLOG is very much a personal computer language. Although its syntax (that is, its word order) may seem a little strange at first, as you work through the examples in this book, you should become familiar enough with the language to make it your own and use it for your own personal applications. PROLOG can be used for all sorts of purposes. You can describe absolutely anything: from the view outside your bedroom window, to how to survive in a thunder-storm! You can describe anything, question it, and then do things with it, according to your own ´rules´. You can either write a simple record of addresses and telephone numbers or invent an interactive problem-solving program, such as to find out whether you have a high or low calory content in your cooking (Weight-watcher program), or write a home doctor program that advises you on what to do if you have a headache (Personal-Doctor program).



Looking at the picture of the Purple Planet, you can use PROLOG to describe what is taking place on the planet. But before you begin, let us look at the story so far.

"The spaceship Galactic Enterprise has been in space for one year. It needs a special ore to refuel its nuclear motors. The crew also needs food. The pilot-computer discovers a nearby planet. The spaceship lands. The entire planet is covered with purple lichen. There is no-one to be seen. Captain Tee, Selina and Doc go exploring. They find a cave which leads to an underground city. The city is inhabited by robots. They meet a friendly robot called Zorg. In the city they find the ore thay they need for the ship´s motors. Food and water however are to be found on the surface. This is a problem, since the surface is inhabited by monsters called Nurks."

Atomic sentences (binary form)

The simplest sentence that can be written in PROLOG is called an atomic sentence. ´Atomic´ is used here to mean ´basic´. PROLOG descriptions are made up of basic or atomic sentences which like building blocks are used to make up the language. One way to write an atomic sentence is to use the form

Name of Individual - Name of Relation - Name of Individual

I shall abbreviate this to Ind - Rel - Ind. This type of atomic sentence consists of an individual word linked to another individual word by a relation word. Thus for example in the sentence

Zorg likes Selina

the individual word **Zorg** is linked (related) to the individual word **Selina** by the relation word **likes**.

Again in the sentence

Owls belong-to Land-birds

Owls (individual word) is related to **Land-birds** (individual word) by the relation word **belong-to.**

Here are more some examples of atomic sentences

Ind	Rel	Ind
Zorg	lives-in	city
water	found-on	surface
Nurkl	lives-on	surface
fuel	found-in	city

There are other forms of sentence, but these you will meet later. For the moment however, let us look at some more descriptive ´atomic´ sentences which describe the Purple Planet and which again use the form Ind - Rel - Ind.

Planet	covered-with	lichen
Spaceship	lands-on	Purple-Planet
Capt-Tee	likes	underground-city

These pieces of data can be fed into the computer by using the command word **add,** and by placing the atomic sentences in brackets. For example to add the sentence **(Zorg lives-in city),** you type

```
add(Zorg lives-in city)
```

and press the ENTER key. This fact or piece of data about Zorg is now stored in the computer´s memory.

Here are the other sentences for you to add to the program:

```
add(water found-on surface)
add(Nurkl lives-on surface)
add(fuel found-in city)
add(Planet covered-with lichen)
add(Capt-Tee likes underground-city)
add(Spaceship lands-on Purple-Planet)
```

Note the use of the hyphen in the relations

```
lives-in     found-on
lives-on     found-in
lands-on
```

and in the names of individuals

```
Capt-Tee          Purple-Planet
underground-city
```

Two or more words for the name of an individual or a relation have to be strung together by hyphens. Thus you could have

Ind	Rel	Ind
Chief-Nurk	likes-eating	lost-travellers
You	must-use	hyphens-in-PROLOG

In these two examples, individual words are made up of two or three words strung together by hyphens.

If you have managed to enter these sentences without any difficulty, I would suggest you practise making up your own atomic sentences, using the binary form Ind - Rel - Ind with the following vocabulary from the Purple Planet. You can of course use any words you like. Don´t forget however to use hyphens when necessary.

Ind	Rel	Ind
Captain-Tee	likes	food
Selina	hates	water
Doc	lives-in	fuel
Zorg	lives-on	city
Astra	found-in	surface
Zenith	found-on	
X-101	dangerous	
Chief-Nurk	friend	
Nurkl	not-friend	
Nurk2	attacks	
robot		

Correcting mistakes

Before you press the ENTER key, you can correct any typing mistakes by using the DELETE key (CAPS SHIFT with 0) to delete back to before the mistake, or you can use the cursor keys. To use these keys, press CAPS SHIFT and 5, until the cursor (that is, the flashing L) is just after the letter or word you wish to correct. Now press CAPS SHIFT and 0 to delete your mistake. Of course you can position the cursor by using the CAPS SHIFT key with 8. The directions of the cursor are shown above these keys in the form of arrows. (For further information, you should refer to your Spectrum manual). When you are satisfied that your data (sentence) is correct, press ENTER as before.

Syntax (word order) errors

If you type the word order of your sentence incorrectly (that is, use a form of sentence that the computer does not understand) you will receive an error message which states that your sentence is not a valid sentence.

If you receive such a message, check your sentence for these common errors

1. You forgot to use hyphens (when necessary).
2. You typed **ADD** instead of **add**.
3. You forgot to use brackets.
4. Your word order is incorrect.
5. You forgot to LOAD SIMPLE.

(For a list of error messages see Appendix 1)

save and **load**

After having written your own sentences, you might like to give them a program name and save them on cassette. To do this:

1. Type **save** followed by the name of your program. For example, **save mystuff** or **save MYSTUFF** (the use of capital letters for your program name is optional).

   You can use almost any name you like for your program, as long as you do not use an in-built PROLOG word (such as **add** or any of the others in this book) or a relation word from your own sentences.
2. Press the ENTER key.
3. Press the record and play buttons on your cassette player and record your work.

When the recording is complete the prompt symbol &. will reappear on the screen. (Don´t forget to unplug the connection to the ear socket on your Spectrum when you are recording. If you forget, you may get some

feedback from the cassette player, which will spoil the recording.)

To reload your work
1. Type **load** and the name of your program, namely **load** mystuff.
2. Press ENTER.
3. Press the play button on your cassette recorder. The prompt symbol &. will reappear when your program has been loaded.

Be careful when reloading your own programs. If you mistype the name of your program, for example

load mestuff instead of load mystuff

the computer will obediently search for the misspelt name. All you can do is type **CLOSE** and reload your program.

Programming bugs
As you run some of your programs in PROLOG (or in any other computer language), you are bound to discover mistakes or bugs in the programs themselves. Don´t worry! Bugs or mistakes are a normal part of programming. Sometimes these bugs in your programs are very simple ones, but they are nevertheless frustratingly elusive. It takes time and patience to find them, but find them we do in the end! It is almost impossible to write a perfect program at the first attempt. Good programming in fact is not necessarily writing bug-free programs, but finding your bugs and correcting them as and when they occur. If you think you have problems, then remember that a system´s bug in one of the NASA space launches cost the United States Government billions of dollars. The bug? A missing hyphen!

Points to remember
1. Data or sentences can be written in the form Ind - Rel - Ind. These are known as atomic sentences.
2. Data or sentences are added to the database by means of the command word **add** and by placing your data in brackets.
3. To save your program use **save** followed by the name of your program.
4. To load your program use **load** followed by the name of your program.
5. Don´t forget to LOAD SIMPLE.

Looking at sentences in PROLOG so far, you might be tempted to think that they are really ordinary English sentences. They are not! Although in either English or PROLOG you can write

```
Zorg likes Selina
Nurkl hates Doc
```

it is not correct English to write

```
Nurkl hates visitor
```

unless of course you are using ´headline´ English such as you would find in a daily newspaper, for example

```
Man bites dog!
```

or

```
Jury frees pig!
```

A logical language

PROLOG is a logic-based language. This means that the computer follows a set pattern such as Ind - Rel - Ind, together with rules that you the programmer have invented. You cannot for example say in PROLOG

```
add(Doc eats too-much and would-like a-rest)
```

You would have to use two sentences in accordance with the Ind - Rel - Ind pattern that the computer understands (see Chapter 2). If you try typing in this data statement, you will receive the error message

```
Not a valid sentence form
```

But some of the data in the vocabulary of the Purple Planet does not make sense when you try to make certain binary sentences. The following sentences

Ind	Rel	Ind
Zorg	dangerous	city
Astra	friend	fuel
food	not-friend	Doc

may be acceptable to the computer, but they are not acceptable to us! To use the words, **dangerous**, **friend**, and **not-friend**, you need another form of atomic sentence.

But all the same try typing

```
add(food not-friend Doc)
```

Does the computer accept this sentence?

Building a database

When you describe the planet in PROLOG, you can use simple atomic sentences of the Ind - Rel - Ind form and/or another type of atomic sentence which uses the Ind - Rel form.

Binary relationship

The Ind-Rel-Ind form of sentence which you have met already is also known as a binary relationship. The word binary means two and refers to a relation between two individuals.

Individual	Relationship	Individual
Doc	likes	food

In this example, **likes** is the relation between **Doc** and **food.** The complete sentence - **Doc likes food** - is called a binary relationship.

Unary relationship

Individual	Relation
Astra	friend

The word unary means one and refers to a single relationship word that defines a single individual:

```
Doc visitor
Nurk2 monster
```

Unlike a binary relationship, a unary relationship tells you what category something belongs to. It is like a label attached to an individual

```
surface dangerous
Zorg    friend
lichen  purple
```

To add these descriptions of the Purple Planet to your database you type

```
add(Doc likes food)
add(Astra friend)
```

The **accept** command

If you have a list of individuals with only one relation (that is, a unary relationship, such as **Astra friend** or **Nurkl not-friend**) you can speed things up by using the PROLOG word **accept** for those relationships that are the same.

We will assume for instance that the names of certain individuals in the database are those of robots from the underground city, and that they are friends. You can enter this data into the database. But before you do this you need to establish the identity of the robots from the list of names. You can either type individually

```
add(Zorg robot)
add(Astra robot) etc.
```

or use the PROLOG word **accept**. This saves you having to retype your sentences several times.

Type

```
accept robot
```

Now place the names of the robots in brackets:

```
(Zorg) (Astra) (Zenith) end
```

When you have completed the list of individuals that are robots, you type another PROLOG word - **end**.

You can do the same for the relation **friend.** Instead of typing

```
add(Zorg friend)
```

you can use

```
accept friend
```

and then list all the friend sentences in your program. This will include the crew of the spaceship as well.

Type

```
accept friend
```

Now list their names individually in brackets.

```
(Zorg) (Astra) (Zenith) (Capt-Tee) (Selina) (Doc)
(X-101) end
```

Checking your program

It is a good idea to check that the computer has in fact accepted your data. You can do this by typing either

```
list
```

or

```
list all
```

When you type **list all**, the entire program you have written will appear on the screen. If on the other hand you only want to look at a particular piece of data, you can use list followed by the particular relation you want.

For example, **list friend**

```
Zorg friend
Astra friend
Doc friend etc
```

Input and Output

Data to be added to your database is also called input. Whenever you type a sentence on the keyboard this is an input to the computer. This applies to computing in general and not just to PROLOG. Subsequently, data that is produced on the screen from the computer is called output.

Here are some sample sentences that illustrate input and output.

```
(Input)   accept visitor
          (Capt-Tee) (Doc) (Selina) end
```

```
(Input)   list visitor
(Output)  Capt-Tee visitor
          Doc visitor
          Selina visitor
```

Sentence patterns

As you write your descriptions or data, you will tend to use a particular form or pattern. It is important that you are aware of this and that you try to maintain a pattern throughout your programs. Maintaining a

pattern helps you to check your programs and at the same time makes the answering of questions more efficient, especially from the computer´s point of view. You could write

```
Zorg friend
Capt-Tee friend
```

or

```
Zorg is-a-friend
Capt-Tee is-a-friend
```

But whichever pattern you use, stick to it! Don´t write for example a mixture of

```
Zorg is-a-friend
```

and

```
Capt-Tee friend
```

The computer will not mind, but you could find yourself in a muddle later as you expand your use of PROLOG (see the section on List patterns in Chapter 3).

Points to remember
1. The sentence form Ind - Rel - Ind is known as a binary (two) relationship.
2. The sentence form Ind - Rel is called a unary (one) relationship.
3. To speed up the input of data having the same relation word, you can use the command word **accept.**
4. It is very important to maintain a consistent sentence pattern when building a database. Don´t mix your sentences. Keep to one form.

Activity 1

Who, What and Where?

From the vocabulary and the story, find out who and what live where. Describe this data and put it into the database.

For example, we know that Nurks are monsters and that they live on the surface. So we could write

```
accept dangerous
(Chief-Nurk) (Nurkl) (Nurk2) (surface) end
```

Where do you find food and water or fuel for the spaceship? Who likes whom?

Remember to use both binary and unary sentences, Ind - Rel - Ind and Ind - Rel. Use **add** and use **accept.** Invent your own words. Change the story if you like. When you have completed your database, type **list all** and check to see whether the data that you have compiled is correct or not.

Editing in PROLOG

We have already discussed how to correct typing and spelling errors using the cursor keys. But sometimes you may press the ENTER key before realising that you have made a mistake, or you may later decide to make changes in a completed program. For this reason PROLOG has an in-built editing facility. Often you will need to delete certain words or change a whole sentence. You have several choices for editing your program.

1. **delete.** This erases incorrect sentences by referring to their relation (Rel).

To use **delete ,** you must first list the sentences that relate to the one you wish to remove. For example, type

```
list friend
```

The screen output is

```
Zorg friend
Astra friend
Zenith friend
Selina friend
Doc friend
Capt-Tee friend
```

To remove **Astra friend,** you type

```
delete friend 2
```

Now type **list friend,** to check that Astra has been deleted.

To add new sentences just use **add** in the normal way, that is

```
add(Spectrum friend)
```

or use **accept friend** and place the new data in brackets.

Alternatively you can type

```
delete(Astra friend)
```

In this case, **delete** is used as the opposite of **add.**

2. **kill.** You can remove all the data about friend by typing

```
kill friend
```

3. **kill all.** More drastically you can destroy the entire program by typing

```
kill all
```

4. **edit.** This allows you to alter sentences in your database. For example

```
list found-on
```

The screen output is

```
food found-on surface
water found-on surface
Chief-Nurk found-on surface
Nurk1 found-on surface
Nurk2 found-on surface
```

If you wish to change food in sentence 1 to lichen, you type

```
edit found-on 1
```

The screen outputs

```
1 (food found-on surface)
```

The 1 indicates the first sentence in your list of sentences. You can now alter this sentence by using the cursor controls (CAPS Shift with 5) and (CAPS Shift with 8) in conjunction with the delete key (CAPS Shift with 0). To delete the word ´food´ you use CAPS Shift with 0. You then position the cursor and type in lichen. You can also alter the order of your sentences by changing the number before the sentence on the screen. Thus to change the position of (**lichen found-on surface**) change 1 to another number.

From English to PROLOG

A small vocabulary in PROLOG can be used to express a wide range of ordinary English sentences.

Capt-Tee is-taller-than Selina

also means that Selina is shorter than Capt Tee.

Doc is-older-than Capt-Tee

also means that Capt Tee is younger than Doc. The computer however cannot read meanings into sentences that are not explicitly stated. It literally accepts what you tell it. PROLOG in fact is very precise, as the following nonsense sentence will demonstrate.

Nurkl gorms-up fetter

This is quite acceptable in PROLOG, and as far as the computer is concerned is a true statement. In ordinary English this sentence could have several meanings. It could mean that Nurkl is eating (gorms-up) a particular kind of cheese (fetter) or it is perhaps propelling itself up a certain hill called a fetter.

The computer does not interpret sentences, it only knows what you tell it.

Logical nonsense

PROLOG stands for Programming in Logic. You may have come across formal logic at school, either using logic blocks and tracks, or as a word exercise in mathematics. Logical statements do not necessarily have to make sense in English. They do however have to obey the rules of logic. This you have already come across in PROLOG. You have been using the binary relationship Ind - Rel - Ind and the unary relationship, Ind - Rel. If you apply these forms or patterns to the following data, you can produce some quite extraordinary sentences.

Individual	Relationship
I	likes
Selina	is-afraid-of
Doc	helps
Nurkl	
Nurk2	

Here are some sentences in ordinary English and their equivalent in PROLOG

```
English:  I help Selina and Doc
PROLOG:   I helps Selina
          I helps Doc
```

In the above example you need two atomic sentences for one in English. Note also that the spelling of the relation word does not change!

```
English:  Doc is helped by me
PROLOG:   I helps Doc
English:  Selina likes me
PROLOG    Selina likes I
English:  Selina is afraid of Nurkl and Nurk2.
PROLOG:   Selina is-afraid-of Nurkl
          Selina is-afraid-of Nurk2
```

Asking questions (atomic questions)

Having established your database about the Purple Planet, you are now ready to ask questions.

The simplest form of question is an atomic question using the PROLOG word **is.** Atomic questions using **is** are written in exactly the same way as atomic sentences. They are always enclosed in brackets:

```
is (Capt-Tee visitor)
```

The computer scans the database to see if it can find the sentence that you are querying. If it does, it outputs YES. In this case the answer is YES.

In the next example

```
is(Nurkl friend)
```

the computer does not find this sentence in the data base, so the answer is NO. What the computer does, is to match your input sentence with what is stored in the program. If the sentence can be matched with a sentence already in the program, then the output is YES.

Try asking these questions

```
is(food found-on surface)
is(Chief-Nurk dangerous)
is(Zenith robot)
is(Zorg lives-in city)
is(X-101 lives-on surface)
is(fuel lives-on surface)
is(Nurkl gombles slithy)
```

A query game

You might like to try this with your friends. First create your database and then get your friends to ask questions about the planet. If they receive a YES to their query, they score a plus point. If the answer to their query is NO, then they are awarded a minus point. Working with young programmers, I have found this game great fun. It all depends of course on what you have put into your programs in the first place!

There are no fixed rules on how to play the game, but here are a few suggestions.

1. Tell them a few things about the planet and then ask a friend or group of friends to identify the characters in the story. You must provide them with the basic vocabulary and tell them how to query the database. The players have to use is.

(a) Who is Zorg? Is he or she a visitor, monster or robot? Ask a question such as

```
is(Zorg visitor)
```

The answer to this query is NO, so they receive a minus point. On the other hand if they ask

```
is(Zorg robot)
```

they receive a plus point and can go on to the next question or part of the game.

2. Having established who the characters are, the players can now find out more details about the characters.

(b) Where does Zorg live? In the city, or on the surface?

```
is(Zorg lives-in city)
```

(c) Is Zorg a friend or enemy?

```
is(Zorg friend)
```

3. Another approach is for the players to land on the planet and assume the identity of the space visitors.

(a) You land on the planet and look for fuel and food supplies. Is the planet a friendly place or is it full of dangers? You can either run into a monster (score a minus point) or meet a friend (score a plus point). This you do by asking questions such as

```
is(Zenith friend)
```

The answer is YES, so you score a plus point. But if you ask

```
is(Nurk2 friend)
```

the answer is NO and you score a minus point.

You can meet the following:

Chief-Nurk	Zorg
Nurk1	Zenith
Nurk2	Astra

Can you survive on the planet by finding food and water? Can you make friends, and at the same time avoid the monsters? Make up your own questions and rules to play this game. Test your friends and perhaps offer them a few surprises.

Database for the Purple Planet

Here is a database for the planet created by pupils at Whetley Middle School in Yorkshire.

Characters

Capt-Tee	visitor	Chief-Nurk	monster
Selina	visitor	Nurk1	monster
Doc	visitor	Nurk2	monster
Zorg	robot	Nurks	monster
Astra	robot		
Zenith	robot		
X-101	computer		

Friend and foe

Zorg	friend	Chief-Nurk	not-friend
Astra	friend	Nurk1	not-friend
Zenith	friend	Nurk2	not-friend
Doc	friend	Nurks	not-friend
Selina	friend		

Who lives in the underground city?

```
Zorg   lives-in city
Astra  lives-in city
Zenith lives-in city
```

Who lives on the surface?

```
Chief-Nurk lives-on surface
Nurk1 lives-on surface
Nurk2 lives-on surface
Nurks lives-on surface
```

What is found on the surface

```
food  found-on surface
water found-on surface
```

What is found in the city?

```
fuel found-in city
```

Extra data

```
Chief-Nurk  attacks city
Nurk1       attacks city
Nurk2       attacks city
Nurks       attacks city
Chief-Nurk  dangerous
Nurk1       dangerous
Nurk2       dangerous
Nurks       dangerous
surface     dangerous
```

Using PROLOG now!

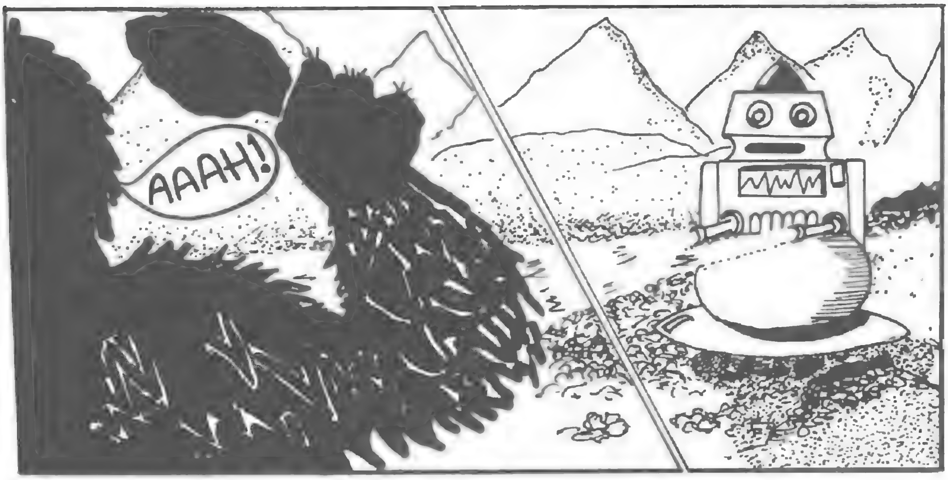
The applications of PROLOG for building an active database are limitless. You should start now. Build up a database of subjects or things that interest you, or if you prefer, just experiment with the language and see what you can do with it. It is up to you. Don´t just read this book, but switch on your Spectrum and start typing! As you work through the sections of this book, trying out some of the ideas and in particular ideas of your own, you will learn how to expand your databases and, more important still, how to manipulate them and make them do what you want them to do!

Here are just some of the ideas for getting started, suggested by young PROLOG users.

Aircraft spotting
Chinese cooking
Sports - Olympic records
Football results
Personal record
Gardening calendar
Family Tree
Athletic chart
Stamp collection
First Aid instructions
Weather record
Historical facts
Joke Box - a collection of favourite jokes and gags

Whatever your interests, you should find PROLOG a useful tool in the pursuit of a hobby or a specific subject you wish to study.

2 The Nurks Strike Back!



Variables in PROLOG

Asking questions so far has been rather a slow process. If you want to find out who lives on the surface for example, you have to work through each individual name until you strike lucky, or simply list the relation. There is however a quicker method. This is to use what are called variables. You have probably already come across variables in Maths. They are simply letters which are used in the place of missing numbers, for instance

```
x + 10 = 20
```

where x is a variable standing for the numbers that added to 10 equal 20, or

```
x - y = 15
```

where x and y are variables standing for the numbers that in a subtraction sentence equal 15.

Finding the unknown

A variable in PROLOG stands for anything or any one who is unknown in your question. It is a kind of place holder, waiting to be filled with a value (that is, the name of the missing thing or person). Thus, to find out if there is any one who lives on the surface of the planet, you type

```
is( x lives-on surface)
```

This means, is there any one x, who lives on the surface? Since the Nurks live on the surface of the planet, the screen output for this query is YES.

In PROLOG we use the letters **x,y,z,** or **X,Y,Z** as variables. We can also use **x,y,z,** or **X,Y,Z** followed by a number such as **x1,x2** or **y2,y3** etc.

For example you could have

```
is(X22 lives-on surface)
```

or

```
is(Y9 lives-on surface)
```

Try some of these letters with numbers yourself. How high can you go?

Although you can only use x,y,z or X,Y,Z, it does not matter which letter you choose to use. Try asking

```
is(y lives-on surface)
is(Z lives-on surface)
is(X lives-on surface)
is(z lives-on surface)
```

The answer will always be the same.

Here is another question to find out if there is anything in the underground city:

```
is(x found-in city)
```

From the database, we know that rocket fuel is to be found in the city, so the screen output is YES.

Here are another two examples

Is somebody a friend?

```
is(x friend)
```

Answer

YES

Does Astra live somewhere?

```
is(Astra lives-in y)
```

Answer

YES

Using two variables

You can ask a question using two variables, for instance

```
is( x lives-on X)
```

The answer to this question is YES.
What you are asking in plain English is - does somebody live on something? The little **x** stands for the names - **Chief-Nurk, Nurkl, Nurk2** - while the big X stands for the word **surface.** Now try asking

```
is(X lives-on x)
```

This time the big X stands for the names of the monsters, while the little x stands for the word **surface.**

However if you ask

```
is( x lives-on x)
```

the answer will be NO. Why?
In this question we have asked whether someone lives on someone, that is - Does Chief-Nurk live on Chief-Nurk...? The computer searches for the relation ´lives-on´ and gives the x variable the value Chief-Nurk, Nurkl or Nurk2. Then it starts looking for the second x in our question which is in fact the same as the first x. In fact it is looking for

```
Chief-Nurk lives-on Chief-Nurk
Nurkl lives-on Nurkl
Nurk2 lives-on Nurk2...!
```

Once you have used a particular variable for a name, the computer will keep that name for the rest of the question.

The same rule applies to y,z and Y,Z:

```
is(y lives-on Y)
```

or

```
is(Z lives-on z)
```

The answer to both these questions is YES. However if you ask

```
is(z lives-on z)
```

the answer will be NO.

Points to remember
It is a good idea to experiment with these PROLOG variables, until you have understood how they can be used. Remember that they are place holders, which represent the unknown in your questions. Variables are in fact used in everyday life. You are probably familiar with the TV commercial which asks housewives which margarine is creamier - Brand X or Brand Y?

We could add this information to the data base and ask

```
is( x creamier)
```

but I am afraid the answer would not be very helpful. It would always be YES!

Using - **which**

The PROLOG word **is** provides us with the following facts:

(a) Someone lives on the surface of the planet.
(b) Something is found in the underground city.
(c) Somebody is a friend.
(d) Someone lives in the city.
(e) Something is found on the surface.

It would be more useful on the other hand to know who and what lives on the surface, in the city, etc. This we can find out by using another PROLOG word, **which**.

Let us ask the same questions again, but this time using **which**, instead of **is**.

```
which(x:x lives-on surface)
```

The answer to this query is

```
Chief-Nurk
Nurk1
Nurk2
No (more) answers
```

```
which(x:x found-in city)
```

```
Answer
   fuel
   No (more) answers


      which(x:x friend)

Answer
   Capt-Tee
   Selina
   Doc
   Zorg
   Astra
   Zenith
   No (more) answers
```

But we have used x twice and a colon(:) as well! Whenever you use the word **which**, the sentence that follows is always composed of two parts.

In the first example,

which(x:x lives-on surface)

the first part is x followed by a colon (:). The second part is **x lives-on surface.** (In this example I have used x. I could have used z or y, or X or Y, it does not matter which letter you choose.)

Answer pattern

The first part is called the answer pattern. It is separated from the second part by a colon (:).

Data pattern

The second part is called the data pattern. This second part is your question

```
x lives-on surface
```

The data pattern must have the same form or pattern as the sentences in your database. This is because PROLOG works by matching patterns.

But why have the first part? Why not just write

```
which(x lives-on surface)
```

To understand why the first part is necessary, let us examine what the computer does. The computer scans the database, searching for the data sentence

```
(x lives-on surface)
```

It finds three sentences to match the data pattern in your question

```
(1)  (Chief-Nurk lives-on surface)
(2)  (Nurk1 lives-on surface)
(3)  (Nurk2 lives-on surface)
```

The x of your question is the x in

```
(x lives-on surface)
```

so the computer outputs

```
Chief-Nurk
Nurk1
Nurk2
```

and since there are no more sentences to be matched, it outputs

```
No (more) answers
```

The variable box

In order to output your query, the computer uses an answer pattern, which you have also labelled x. The answer pattern is like a box. When you type the first part of the sentence, you create a box labelled **x**, which waits to receive the answer to the x of the second part of the sentence. The following illustration will help to make this clearer.

First part of sentence   Second part of sentence

The answer to **(x lives-on surface)** is placed in the box labelled **x**. This all happens of course behind the scenes. All you see on your screen is the answer to **(x**

lives-on surface), that is, the second part of your question or data pattern.

Let´s try another query, using y this time instead of **x**:

```
which(y:y found-in city)
```

The first part of the sentence sets up a box labelled y. The second part, (y **found-in city)**, tells the computer to search for this sentence in the database. The answer is

```
fuel
No (more) answers
```

The output - **fuel-** is placed in the y box.

Expanding the variable box

You can also play with the box and use it to output extra data. Here are some more questions

```
which(z:z lives-in city)
```

Answer

```
Zorg
Astra
Zenith
No (more) answers
```

```
which(y:y dangerous)
```

Answer

```
Chief-Nurk
Nurk1
Nurk2
surface
No (more) answers
```

Let´s now expand the variable box:

```
which(z is a friendly robot:z lives-in city)
```

Answer

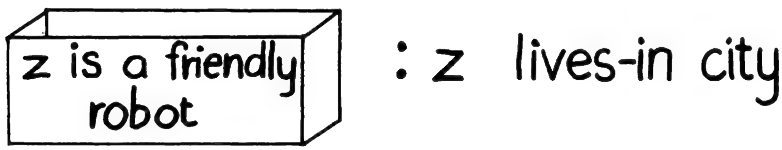
```
Zorg is a friendly robot
Astra is a friendly robot
Zenith is a friendly robot
No (more) answers
```

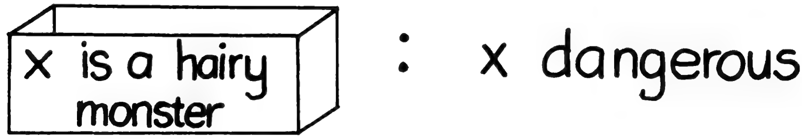
```
which(x is a hairy monster:x dangerous)
```

Answer

```
Chief-Nurk is a hairy monster
Nurk1 is a hairy monster
Nurk2 is a hairy monster
surface is a hairy monster(oops!)
No (more) answers
```

In these last two examples we have created two boxes, z and x. We have also added additional data to the boxes, before they have been assigned values.





Whatever you write in the first part of your question will be printed on the screen, together with the answer to your data pattern.

Try these atomic questions

```
which(what a lovely y:y dangerous
which(rocket z:z found-in city)
which(z is good for you:z found-on surface)
which(y:y friend)
which(X:X likes Selina)
which(Y:Y lives-on x)
which(Z: Chief-Nurk hates Z)
which(x helps y:x likes y)
```

In a question with several variables, you only get the names of individuals referred to in the variable box. For instance

```
which(z:z hates y)
```

You do not get the word - **visitor** - because the unknown y of the data pattern is not given in the answer pattern (that is, our box). In other words you didn´t ask for it. You only asked for z.

```
which(x:y lives-on X and y hates x)
```

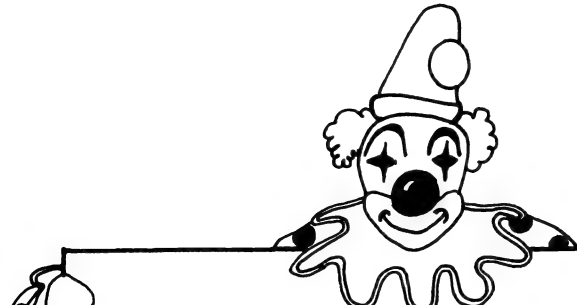
Here you ask for the value of **x**. The answer is therefore - visitor. The other variables y and X do not appear in the answer pattern (that is, the first part of your question) and therefore they are not retrieved from the database.

However you can ask

```
which(x helps y:x likes y)
```

and receive two names for **x** and for y respectively, because you included both **x** and y in the answer pattern (that is, the first part of your question). Try this and see for yourself.

There are various combinations of text and variables that you can use. Try your own. How many variables can you use? How much text can you print in the variable box?



Cracker jokes

You can either write your own cracker jokes or collect them from Christmas crackers and add them to the Joke Box suggested at the end of Chapter 1.

Database

```
add(runner-beans help athletes)
add(hippopotamouse is-largest-type mouse)
add(an-astronut is-called a-crazy-spaceman)
add(a-safe-robbery is-an-easy robbery)
```

Questions

```
which(x:x help athletes)
which(x:x is-largest-type mouse)
which(x:x is-called a-crazy-spaceman)
which(y:y is-an-easy robbery)
```

Molecular questions

You may want to ask different kinds of questions, using a more concise form than the atomic sentence. This can be done by combining atomic sentences into molecular sentences. The simplest form is

```
Ind - Rel - Ind and Ind - Rel - Ind
is(Zorg likes Selina and Zorg likes Doc)
```

What you have here in fact are two atomic sentences that are joined together by the conjunction **and**. (Alternatively, the symbol **&** can be used.) Whereas in English you can say

Zorg is a robot and likes visitors

in PROLOG you must use two atomic sentences to say this. The second half of the sentence in ordinary English must be replaced with another atomic sentence in PROLOG. The two atomic sentences are joined together by **and**.

```
is(Zorg robot and Zorg likes visitor)
```

In this example you must repeat the name Zorg.

Here are some more examples

```
is(food found-on surface & water found-on surface)
is(x found-in city and x likes y)
```

You might want to ask whether the Chief Nurk hates visitors and attacks them:

```
is(Chief-Nurk hates visitor & Chief-Nurk attacks
visitor)
```

In all these examples the pattern Ind - Rel - Ind has been maintained throughout.

Let us leave molecular questions for the moment and return to our story.

Extending the game

"The visitors, alias the crew of the spaceship, led by Captain Tee, make their way to a water hole on the surface. They are accompanied by Zorg and Astra. They arrive safely at the water hole and begin to fill their containers. Suddenly a Nurk appears on a ridge opposite. Zorg responds by firing a laser torch at the monster. The Nurk roars with pain and anger. It retreats over the top of the ridge. The crew finish filling their containers and return to the underground city. The Nurk however has only had its fur singed. It follows the crew and discovers the entrance to the cave. Later that evening the Nurks, in full force, attack the city."

Activity 2

Here is some more data

Individual	Relationship
laser-torch	strike-back
Nurks	retreats
fur	follows
containers	is-singed
pain	fill
city-entrance	fires
crew	roars-with
water-hole	takes-note-of
	go-to

Write a database for the following questions.

```
is(crew go-to water-hole and crew fill containers)
is(Zorg fires laser-torch and Nurkl roars-with pain)
is(fur is-singed and Nurkl hates robot)
is(Nurkl follows crew and Nurkl takes-note-of
city-entrance)
is(Nurks strike-back and Nurks attacks city)
```

Molecular questions with variables
You can query these sentences, using **is** and **which**:

```
is(x is-singed and y hates robot)
```

Answer
YES

```
is(z go-to water-hole & z fill containers)
```

Answer
YES

```
is(y follows z and y takes-note-of x)
```

Answer
YES

This question in ordinary English is - Does somebody follow somebody else and does that somebody take note of something?!

```
which(x:x fires laser-torch and y roars-with
pain)
```

```
Answer
   Zorg
   No (more) answers
```

```
which(y:z follows crew and z takes-note-of y)
```

```
Answer
   city-entrance
   No (more) answers
```

Joke Box with molecular questions

Data

```
add(gooseberry green-and-hairy and gooseberry
goes-up-down in-lift)
add(pavement wears shoes and pavement has-no
feet)
```

Questions

```
which(z:z green-and-hairy and z
goes-up-down in-lift)
which(X:X wears shoes and X has-no
feet)
```

One by one

Sometimes you may require only a single answer to your question and for this you can use the command word **one**. This is used in the same way as **which**, but with a difference. Instead of the computer giving you all the possible answers to your question, it outputs only one answer at a time.

```
Zorg likes Doc
Zorg likes Capt-Tee
Zorg likes visitor

one( x:Zorg likes x )
```

Answer

```
Doc
more? (y/n)
```

You receive one answer followed by the prompt - **more? (y/n)** If you answer yes, the next answer is given.

```
Capt-Tee
more? (y/n)
```

This continues until all the sentences have been matched. The final answer for this example is

```
visitor
No (more) answers
```

The command word **one** is therefore useful for controlling the number of answers you require for a particular question. For example,you might be choosing volunteers for a summer camp. You require two climbing instructors and four swimming instructors. Supposing you have numerous applicants, who are all equally qualified, you can select the required number by using one.

```
(J Sharp) offers swimming
(W Jones) offers climbing
(B Dutton) offers climbing
(F Smith) offers swimming
(G Stringer) offers swimming
(M Woods) offers swimming
(T Betts) offers climbing
(V Walker) offers swimming
```

```
one (x:x offers climbing)
```

```
Answer
   (W Jones)
   more? (y/n)  yes
   (B Dutton)
   more? (y/n)  no
```

```
one (x:x offers swimming)
```

```
Answer
   (J Sharp)
   more? (y/n) yes
   (F Smith)
   more? (y/n) yes
   (G Stringer)
   more? (y/n) yes
   (M Woods)
   more? (y/n) no
```

Activity 3
Use the following new data, together with some words of your own to describe the planet, using molecular questions.

Individual	Relationship
purple-lichen	blows
mountains	grows
hot-wind	rise-above
clear-water	flows
pink-sky	shine
blue-moons	situated
stars	appear
dust	in-the
landscape	fit-to-drink

Here are a few examples

```
is(hot-wind blows and dust blows)
which(x:x rise-above mountains and x shine)
which(y:y flows south and y fit-to-drink)
one( z:z shine in-the pink-sky)
```

Summary

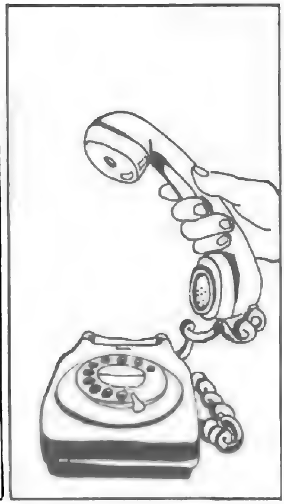
In PROLOG there is no distinction between program and database. To write a program means to create a database. Data is added to the database by writing descriptive sentence. These sentences are made up of two kinds, binary form and unary form. These sentences can also be questioned, using the same format by which they were added to the overall program. There are four types of questions:

(1) Atomic questions using **is**.
(2) Atomic questions with variables using **is** and **which**.
(3) Molecular questions using **is**.
(4) Molecular questions using **is** and **which**.

Use **one** to output one answer at a time to your question.

To check your individual data sentences use **list** followed by the relation word. For example **list friend** or **list likes**. To check your entire program, type **list all**.

3 Birds, Friends and Neighbours



A popular television quiz programme called Master Mind tests contestants on items of general knowledge and asks them questions about a specific subject they have chosen. PROLOG allows you to build your own encyclopedia of knowledge, such as would be useful if you were a contestant in Master Mind. For example, you might be interested in birdwatching. Looking at and identifying birds can be a rewarding and fascinating pastime. The British Trust for Ornithology, established in 1962, recruits birdwatchers who keep records of common and rare species of bird throughout the British Isles.

PROLOG is ideal for collecting and storing such information. Already you have the tools to do this. You might store your knowledge something like this.

```
add(owls belong-to birds-of-prey)
add(eagles belong-to birds-of-prey)
add(sparrow-hawks belong-to birds-of-prey)
add(kestrels belong-to birds-of-prey)
```

Lists as individuals

A more convenient way of storing data is to make a list of the birds of prey. This is done by placing the individual names of the birds inside brackets as follows

```
(owls eagles sparrow-hawks kestrels)
```

Instead of writing a single sentence for each bird, I have placed all the birds together in a list, and then used this list in a single sentence. Instead of four separate sentences, we now have just one

```
add((owls eagles sparrow-hawks kestrels)
belong-to birds-of-prey)
```

You will notice that the list of birds is enclosed in brackets, which in turn is placed inside the brackets of the data sentence.
As soon as you place the names of the birds in brackets, you have created a list, which the computer accepts not as separate individuals, but as one individual (word), just as it accepts an individual like Zorg or Capt-Tee.
As far as the computer is concerned, the list, however long it may be, is one individual.
Here are some more examples of the use of a list as an individual

```
(cormorant puffin albatross) belong-to sea-birds
(thrush blackbird lark) belong-to songbirds
```

Everyday lists
We use lists all the time in everyday situations such as shopping or writing lists of things to do, or compiling a list of team members in the school football team. Lists are, for most of us, the normal method of recording things. In PROLOG all you have to remember is to use brackets to create your lists, and to remember that the computer accepts a list as an individual.
Let´s look at some more bird examples using lists. The following examples are classified by colour. Here as always it is important to keep to a set pattern. It will help you to read the data as well as help you to spot any errors in your program.

Binary form

Individual	Relation	Individual
(rook crow raven blackbird swift)	belong-to	Black-land-birds
(owl skylark thrush eagle warbler)	belong-to	Brown-land-birds
(pigeon cuckoo coal-tit sparrow-hawk)	belong-to	Grey-land-birds
(robin bullfinch chaffinch swallow)	belong-to	Red-breast-or-head

Using two lists

```
(duck swan) nest-site (marshland river-side)
(house-martins swallows) nest-site (eaves roofs)
```

Unary form

Individual	Relation
(Dodo Moa)	extinct
(chicken turkey duck goose)	domestic

When using a list or a number of lists, always check your use of brackets. PROLOG will in any case prompt you, should you leave out any final brackets. Try typing

```
add((robin thrush) belong-to songbirds
```

The computer will prompt you with a figure 1. This stands for the missing bracket.

List processing

Using lists in PROLOG is known as list processing. It is a very versatile tool for organizing and manipulating data. There are various combinations of lists that you can use. Some of these will be discussed later. For the moment let us look at data sentences that use one or more lists.

Identifying birds in flight

1. By shape of wings

```
(swallows swifts) have-wings (long pointed)
(doves pigeons) have-wings (broad pointed angled)
(finches sparrows) have-wings (short)
```

2. By flight patterns

```
(birds of prey) flight-details (hovers
broad-long-wings solitary-flight)
(gulls) flight-details (slow-wing-beats glide
fly-in-flocks)
```

List patterns

Maintaining a pattern is very important. In the previous examples the pattern was

```
(type of bird) flight-details (details)
```

or

```
(name of birds) have-wings (details)
```

Each description used this pattern, that is, a list - relation word - list.

We could have written

```
gulls flight-details (slow-wing-beats glide
fly-in-flocks)
```

that is, ´gulls´ without the first set of brackets. In this case, the pattern would have been interrupted.

(gulls) is therefore written as a list and not as an ordinary individual. The same applies to **(short)**.

The empty list

This idea of pattern is again important should you wish to use an empty list in your database. For example you might compile a database of birds that swim, in which you would like to include some that do not fly:

```
(arctic-tern) flight-details (hover-above-waves
fly-in-flocks)
(penguin) flight-details (  )
```

Penguins of course do not fly, so the second list is empty. Writing ( ) however does not mean nothing! It is known as an empty list and is a useful device in the database.

Changing the subject for the moment, an empty list could be used in a family record for example

```
The Makepeace family 1735-1812
(James Katherine) parents-of (May Thomas)
(Kevin Sarah) parents-of (Alice Jane)
(Brian Wendy) parents-of (Susan)
(Edward Mary) parents-of ( )
```

Note that there is a consistent pattern throughout the sentences. Each sentence is composed of two lists, one for the parents and one for the children. At a glance it can be seen that Edward and Mary have no children. The empty list has been used to indicate this fact.

Prose writing using lists

Another feature of lists is the ability to record information in the form of ´prose´, that is ordinary English sentences.

```
Unusual sightings
   (gold-crest) located (15th Jan 1984 in garden in
   Warwick Avenue, London)
   (white-bird) located (16th Jan 1984 in West
   Hampstead - an albino blackbird with white feathers)
```

Note that when you are using lists for prose writing, you do not need hyphens, except of course for special emphasis or where they are used in ordinary English sentences. Having said this, you will find that most of the programs in this book have been written using hyphens in order to save on lengthy descriptions in the database. But this by no means implies that you have to be brief or that you have to use abbreviated forms of sentence. You might for example wish to compile a database of quotations, which you would like to retrieve from the data base in full. In this case you would use the full prose form of sentences rather than short data descriptions.

```
(Revelation 6 13) verse (And the stars of heaven
fell to the earth, even like a fig tree casts its
unripe fruit, when she is shaken by a mighty wind)
```

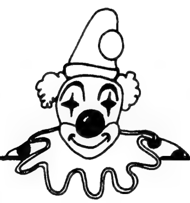
As I have said before, it all depends on what you want to do with PROLOG. What really matters is that your programs do the job you want them to do.

Secret messages

Another use of prose writing in lists could be to record secret information for your friends to find by asking the 'right' questions:

```
Capt-Tee says (Meet me at the big crater, tonight at
2200 hours)
Chief-Nurk says (visitors are making their way to
the big crater. We will attack when it gets dark)
```

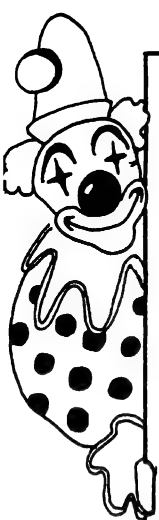
I leave the rest to your own imagination.



Joke Box again

Did you know that most comedy is manufactured? Professional scriptwriters in show business, who have to make a living out of comedy, seldom rely on inspiration alone. They use certain techniques to manufacture gags and jokes. One way to do this is to build up a database of associated words. The database could be a collection of car-words such as

```
engine boot seats glove-compartment
petrol gears etc.
```



Writers first try to find associations between these words and other subjects, and then they apply the technique of exaggeration. Taking size as an example, you could have the following jokes

```
add((Our car is so small, you can only put one
finger into the glove compartment) joke)
add((My aunt´s car is so old that it has grey hair
under the bonnet) joke)
add((And in winter, my aunt puts a thick knitted
sock in its boot) joke)
add((my car is so small, that it never goes without
its rattle) joke)
```

I have recorded the jokes using the prose facility in lists. The prose sentences are the individual, while the relation word is joke. The data is stored in the unary form Ind - Rel.

Suppose for example you would like to write a comedy script for a school show or perhaps for a party; this can be done by creating your own store of words and ready-made jokes. To make a list of car-words you could use **accept car**, and then place each individual word in brackets:

```
accept car
(engine) (bonnet) (glove-compartment) etc.
```

Ready-made jokes can be added to your database using a list. A list allows you to write your jokes in ordinary English:

```
add((what can jump higher than a house? Anything. A
house can´t jump) joke)
add((What is an archaeologist? A man whose career
is in ruins) joke)
add((What is the best time to buy chicks. When they
are going cheap) joke)
```

Best sellers

```
(Ghost stories by Major Jump) best-sellers
(Channel Swimmer by Frances Near) best-sellers
(Cliff Fall by Eileen Dover) best-sellers
(No Hiding Place by I.C. Hugh) best-sellers
```

Use **accept best-sellers** and make up your list of titles.

Points to remember

1. You can create a list by placing your data inside brackets.
2. The list is treated by the computer as an individual.
3. By using lists your data can be entered in the form of ordinary English or prose.
4. Single names can also be stored in your database in the form of a list (gulls), which is sometimes necessary in order to preserve the pattern of your sentences.

   Thus ´gulls´ can be written alternatively as (gulls). The first is a word, the second is a list: both however are individuals.
5. Remember to count your brackets.

Bird-watching record

I find lists ideal for recording sentences of data in place of a single individual name. For example, if I want to record the characteristics of a particular species of bird, I could use a list to do this.

Individual	Relationship	Individual
pigeon	has	(0010110)

Here I have used a list as an individual, namely **(0010110)**.

This particular list is a coding which identifies any species of bird that I am observing. It works like this: each category is represented by either **1** or **0**, in which **1**= YES and **0**= NO.

For example, the first figure stands for the colour black, the second for brown, the third for grey, etc.

```
(colours) black
          brown
          grey
          long pointed wings
          broad, pointed, angled wings
          common species
          rare species
```

Looking at the data for pigeon, the list **(0010110)** means

0 not black
0 not brown
1 grey
0 not long pointed wings
1 broad, pointed, angled wings
1 common species
0 not rare species

Each figure represents a particular characteristic. A figure **1** in the case of the pigeon means, YES the pigeon does possess this characteristc, while a **0** means NO, the pigeon does not have this characteristic.

Here are the characteristics for a swallow

Ind	Rel	Ind
swallow	**has**	**(1 0 0 1 0 1 0)**

The code for swallow means

1 black
0 not brown
0 not grey
1 long pointed wings
0 not broad, angled wings
1 common species
0 not rare

This system can be applied to virtually any subject. It is already used in many businesses like shops, for example to keep records of stocks or to record information about personnel, their age, qualifications, salaries etc. It is very useful for keeping club records or a record of a hobby collection, in fact almost anything.

Athletics meeting

Here is another system using a list, this time for recording events at an athletics meeting.

```
(Jackie Tarrant) events ( 1 0 1 0 )
(Lorraine Simpson) events ( 1 1 0 0 )
(Irina Laski) events ( 1 0 0 1 )
(Rachel Gascoigne) events ( 0 1 0 1 )
(Stephanie Turner) events ( 1 0 1 1 )
(Philip Beckwith) events ( 0 0 0 1 )
(Gordon Mills) events ( 1 1 1 0 )
(Chris Gregory) events ( 0 1 1 0 )
(Tony Bland) events ( 1 0 0 1 )
(Ian Hart) events ( 0 0 1 1 )
```

In this database, the code applies to both ladies´ and men´s events. Again figure 1 = YES (for those taking part in a particular event, while 0 = NO (not taking part in a specific event). For Ian Hart therefore the list (0 0 1 1) stands for

0 not long jump
0 not high jump
1 track relay
1 hurdles

According to his code, Ian is taking part in the track relay and the hurdles race.

Swimming club

The same system can be applied to the records of your local swimming club. At a glance you can, supposing you are the secretary, monitor your members´ swimming skills and other relevant information.

```
(Nick Turner) record-states ( 1 1 1 1 )
(Robin Stuart) record-states ( 0 1 0 0 )
(Wilfred Smith) record-states ( 1 0 0 0 )
(Frances Pauley) record-states ( 1 1 0 0 )
```

The code for Frances Pauley means

1 breast-stroke
1 crawl
0 not more than 15 lengths
0 not life-saver (bronze medallion)

Friends and neighbours

Your computer can also help you to be a good neighbour. The following database, using the same system as for birds, athletics and swimming, is called Community Care or Needy Neighbours. You can compile a list of needy persons in your neighbourhood - people who would appreciate a little care and attention, a chat for example, or some help in an emergency. Names in this record are of course imaginary.

```
(Jim Slater) has ( 0 1 1 0 0 0 )
(Mary Biggs) has ( 1 0 1 1 1 0 )
(Fred Brown) has ( 1 1 0 1 1 1 )
(Jane Smith) has ( 1 1 0 1 1 0 )
(Alice Ginn) has ( 0 1 0 0 0 0 )
```

Let´s take the coding for Jim Slater

0 not elderly
1 handicapped
1 mobile (can get out of the house)
0 not living alone
0 no pets
0 no telephone

Compare Jim´s coding with that for Jane Smith

1 elderly
1 handicapped
0 not mobile (bed ridden?)
1 living alone
1 pets
0 no telephone

For these records I have used two lists linked by the relation word **has**. Note the pattern of sentences throughout the record.

Individual	Relation	Individual
(Fred Brown)	has	(1 1 0 1 1 1)

Data retrieval
Let´s take a look now behind the scenes, as it were, and investigate the very centre of PROLOG. This centre is the making of rules. PROLOG is fundamentally a rule-based programming language. Any sentence in PROLOG that contains variables is a rule.

Up to now you have only used variables in questions:

```
is( x belong-to birds-of-prey)
is( y dangerous)
```

or

```
which(x:x water-bird)
which(y:swallow belongs-to y)
```

But variables can be used in data statements, such as

```
add( x belongs-to y)
add( z have-wings X)
```

which means that something belongs to something else, and that some things have wings of a certain type.

This form of rule, although acceptable to the computer, is of very little practical use so far as we are concerned.

R The conditional rule

For a rule to have real practical value, you need to add a conditional statement as follows

```
x belong-to birds-of-prey if x hunt z
```

Let´s take a real life example from birdwatching:

```
owls belong-to birds-of-prey if owls hunt animals
```

The form of the rule is

```
CONCLUSION if CONDITION(S)
```

In our example, owls belong to birds of prey (the conclusion) provided they hunt animals (the condition). You can of course have as many conditions as you like. I have only used one condition in the rule about birds of prey. To apply this rule, it is necessary to compile some more data:

```
add(owls hunt (mice voles birds beetles))
add(kestrels hunt (mice birds fish))
```

? Asking questions

Having described the feeding habits of owls and kestrels, you can now query the database about birds of prey:

```
which(x:x belong-to birds-of-prey)
```

The computer searches the database and finds the rule

```
x belong-to birds-of-prey if x hunt z
```

The variable **z** represents the list of hunted animals which is **(mice voles birds beetles)** in the case of owls and **(mice birds fish)** in the case of kestrels.

Since owls and kestrels can be substituted for the **x** in the sentence

```
x hunt z
```

the computer assigns the value of **x** to the **x** in the sentence

```
x belong-to birds-of-prey
```

and outputs the answer

```
owls
kestrels
No (more) answers
```

It is very important that you should understand how rules are made in PROLOG. Your successful programs depend upon clear and precise rules. It is better to begin with simple, obvious rules, than lengthy complicated descriptions that stretch your and other users´ logic to the extreme!

The following rules are obvious but nonetheless effective for retrieving information from the database:

1. x is-a carnivore if x only-eats animals
2. x is-a omnivore if x eats animals and x eats seeds
3. x eats sea-food if x lives-near sea
4. x feeds-on fish if x nests-on y and y near coast

Let´s substitute names for the variables in these four rules:

1. eagle is-a carnivore if eagle only-eats animals
2. robin is-a omnivore if robin eats animals and robin eats seeds. (The robin like many birds is an omnivore, eating seeds and worms, grubs etc.)
3. gull eats sea-food if gull lives-near sea
4. puffin feeds-on fish if puffin nests-on cliffs and cliffs near coast

In rule 4, there are two conditions, one for the nesting habitat and one for the site of habitat.

As you build your database you will need to make rules, so that you can cross-reference data as and when you need it. Work through the following examples and see if you can adapt the rules to your own type of program.

```
x feeds-on fish if x nests-on y and y near coast
```

```
add(puffin nests-on cliffs)
add(cormorant nests-on cliffs)
add(gannet nests-on cliffs)
add(cliffs near coast)
```

```
which (x:x feeds-on fish)
```

```
Answer
   puffin
   cormorant
   gannet
   No (more) answers
```

```
x is-a carnivore if x mainly-eats y and y
is-found-in sea
```

```
add(herring-gull mainly-eats fish
add(puffin mainly-eats fish)
add(cormorant mainly-eats fish)
add(gannet mainly-eats fish)
add(fish is-found-in sea)
```

```
which(x:x is-a carnivore)
```

```
Answer
   herring-gull
   puffin
   cormorant
   gannet
   No (more) answers
```

Rules for the Joke Box

Here we can expand our use of rules and use not just one rule but three. I have in fact used here one main rule that consists of two sub-rules.

```
Rule
   x built-upside-down if x feet smell and x nose-runs

Sub-rules
   x feet smell if x runs too-much
   x nose-runs if x has-a cold

Data
   Bill runs too-much       Jill has-a cold
   Susie runs too-much      Bill has-a cold
   Fred runs too-much


      which (x:x built-upside-down)


Answer
   Bill
   No (more) answers
```

Individuals in a list

Not only can you use lists as individuals in your programs

Ind	Rel
(milk bread tea jam)	shopping-list
(The great hairy Nurk)	sings

but you can refer to the individuals within a list. Let´s stop and think about this for a moment.

We know that a list is treated as an individual in the same way that an individual name is treated as an individual. We can write for example

```
Doc
Hairy-Nurk
(Milk bread tea jam)
```

These are all valid forms of individuals in PROLOG.

However, to go a stage further, the individual items within a list can themselves be lists. Take for example

```
(Milk bread tea (jam))
```

This list contains four items, one of which is a list, that is **(jam)**.

Again

```
(England (Dorset(Poole)))
```

Here we have something quite different. Each item here is a list. We have a list called England in which there is a list called Dorset, in which there is a list called Poole.

Items within a list can therefore also be lists themselves and have their own individuals within them and so on. It is rather like those Russian dolls which fit inside each other or like the song about the fly on the wing, the wing on the bird, the bird on the egg...

You can also extract the individual items listed in a list, such as milk or bread etc., in the shopping list.

But to demonstrate this manipulation of individuals inside a list, let us return to the databases which used codes for specific items of data.

If we return to the database called Athletics meeting, it is now possible to illustrate the use of a rule which will enable you to retrieve and manipulate the items within a list.

R

```
1.  x events (long jump) if x does (1|y)
2.  x events (high jump) if x does (y 1|z)
3.  x events (track relay) if x does (y z 1|X)
4.  x events (hurdles) if x does (y z X 1)
```

Heads and Tails

In rule 1, the data sentence can be divided into two parts.

In the first part

```
x events (long jump)
```

x is the variable for a competitor´s name, say Lorraine Simpson or Chris Gregory. (Note that you can use any of the in-built variables available in PROLOG.) In the second part

```
if x does (1|y)
```

you meet a new idea in lists. This is the pattern called heads and tails. A list itself can be partitioned into a head and a tail. This ability to partition a list enables you to retrieve or extract individual items from the list. In **x does(1|y)**, the list **(1|y)** has been partitioned into the head 1 and the tail y. The two have been separated by the sign |.

In rule 2, the list is **(y 1|z)**, in which the head is y 1, while the tail is z. Again in rule 3, the list is **(y z 1|X)** in which **y z 1** is the head, and **X** is the tail. If you relate each variable to the events in the rules, you will see that the figure 1 in the head of each list corresponds to the information in the database of each competitor. For instance

```
                         L H T Hu
(Jackie Tarrant) events (1 0 1 0)
```

From the code, we know that Jackie is taking part in the long jump (L), and the track relay (T). If you look at rule 1 you will see - **x does (1|y)** which means Jackie Tarrant does (long jump).

Again looking at rule 3

```
x does (y z 1|X)
```

means Jackie Tarrant does (track relay). Each figure 1 in the rules corresponds to the code given to each competitor. The first column, that is long jump (L), corresponds to the first figure 1 in the list **(1|y)**. The remainder, or tail of the list, is represented by the variable **y**.

The second column, which is high jump (H), corresponds to the second item or figure **1** in the list **(y 1|z)**. The variable, y this time, denotes a space, while the **1** is followed by the tail or remainder of the list:

```
L H T Hu
1 0 1 0
```

Jackie´s code registers 0 or NO for high jump and 0 or NO for hurdles (Hu).

```
which (x:x events ( long jump))
```

Answer

Jackie Tarrant	Gordon Mills
Lorraine Simpson	Tony Bland
Irina Laski	No (more) answers
Stephanie Turner	

Swimming Club

```
(Nick Turner) record-states (1 1 1 1)
(Robin Stuart) record-states (0 1 0 0)
(Wilfred Smith) record-states (1 0 0 0)
(Frances Pauley) record-states (1 1 0 0)
```

Here are the rules for the swimming club:

```
x can-do (breast stroke) if x record-states (1|y)
x can-do (crawl) if x record-states (y 1|z)
x can-do (>15 lengths) if x record-states (y z 1|X)
x can-do (life saving) if x record-states (y z X 1)
```

```
which (x:x can-do (life saving))
```

```
Answer
   Nick Turner
   No (more) answers
```

Needy Neighbours

```
(Jim Slater) has (0 1 1 0 0 0)
(Mary Biggs) has (1 0 1 1 1 0)
(Fred Brown) has (1 1 0 1 1 1)
(Jane Smith) has (1 1 0 1 1 0)
(Alice Ginn) has (0 1 0 0 0 0)
```

```
x situation (elderly) if x has (1|y)
x situation (handicapped) if x has (y 1|z)
x situation (mobile) if x has (y z 1|X)
x situation(living alone) if x has (y z X 1|Y)
x situation (pets) if x has (y z X Y 1|Z)
x situation (no telephone) if x has (y z X Y Z 1)
```

```
which (x:x situation (no telephone))
```

```
Answer
   Fred Brown
   No (more) answers
```

Bird-watching record

```
pigeon has (0 0 1 0 1 1 0)
swallow has (1 0 0 1 0 1 0)
```

```
X char (black plumage) if X has (1|Y)
X char (brown plumage) if X has (Y 1|Z)
X char (grey plumage) if X has (Y Z 1|x)
X char (long pointed wings) if X has (Y Z x 1|y)
X char (broad pointed wings) if X has (Y Z x y 1|z)
X char (common species) if X has (Y Z x y z 1|Y1)
X char (rare species) if X has (Y Z  x y z Y1  1)
```

As the letter variables are used, beginning in these rules with X, the last variable is a letter with a positive number, that is Y1. If more variables were to be used in your database then the next variables would be Z1, x1, y1, z1, Y2 and so on. char stands for characteristic.

```
which (x:x char (common species))
```

```
Answer
   pigeon
   swallow
   No (more) answers
```

More heads and tails

You can use the heads and tails pattern in ordinary queries or questions, as well as with codes.

```
add((robin bullfinch linnet) belong-to Group-A)
```

If you want to know which is the first bird in the list, you can ask the following question.

```
which (x:(x|y) belong-to Group-A)
```

```
Answer
   robin
   No (more) answers
```

If you want to know which is the second bird in the list, you can ask the following question.

```
    which (x:(y x|z) belong-to Group-A)
```

Answer
```
   bullfinch
   No (more) answers
```
To find out all but the last bird in the list, you can ask the next question.

```
    which ((x y):(x y|z) belong-to Group-A)
```

Answer
```
   (robin bullfinch)
   No (more) answers
```

Telephone and address file

Lists are useful for keeping a file of your friends' telephone numbers and addresses.

You might compile something like the following

```
(Ivan Ivanovich) tel((92 39301)(address 5 Church
St)))
(Kim Davies) tel((92 35322)(address 16 Rose
View)))
(Jack Stephens) tel((92 34421)(address 12 Rose
View)))
(Betty Morgan) tel((92 28845)(address 22 Mount
Drive)))
(Andy Sherwood) tel((94 33211)(address 46 West
Ave)))
(Jane Hanham) tel((94 34377)(address 2 Danes
Ave)))
(Kevin Golden) tel((94 35397)(address 17 Low Lane)))
(Roman Nello) tel((94 37389)(address 11 High St)))
(Jane Walmsley) tel((01 566 3423)(address 34 Wick St
W9)))
(Dorothy Erol) tel((01 584 2341)(address 12 Robinson
Rd SW17)))
(Bashir Khan) tel((01 727 5645)(address 2a Bleeding
Heart Yard EC1)))
```

Don't be put off by the number of brackets! If you look carefully you will see that the program is made up of lists within lists. Using lists in this manner is quite easy with practice.

Asking questions

To query this database you will need some rules

```
x tel London-area if x tel((01|y)z)
x tel Leeds-area if x tel((92|y)z)
x tel Ilkley-area if x tel((94|y)z)
```

These rules use the head and tail pattern. They contain both a telephone number and an address

The telephone number for London is contained in the list (01|y) in which the code for London is 01, the head, while the tail of the list y contains the telephone exchange and the individual telephone number. The address is contained in the list represented by z. Another way of looking at this is to call the list (01|y), and z sub-lists of the main list that contains both, that is ((01|y)z).

Other codes are

```
Leeds 92
Ilkley 94
```

You can query any part of the data by using variables, lists and heads and tails patterns.

Queries

Who has a London telephone number?

```
which(x:x tel London-area)
```

Answer

```
(Jane Walmsley)
(Dorothy Erol)
(Bashir Khan)
No (more) answers
```

What is Jane's surname?

```
which(x:(Jane x) tel London-area)
```

Answer

```
Walmsley
No (more) answers
```

Who lives at 46 West Ave?

```
which(x:x tel(y (address 46 West Ave)))
```

Answer
```
(Andy Sherwood)
No (more) answers
```

What´s the address of Roman Nello?

```
which(y:(Roman Nello) tel(x  y))
```

Answer
```
((address 11 High St))
```

I find that sometimes I know a person´s address, but cannot remember the postal area or postal code.

What is Bashir´s surname and postal area?

```
which((x y):(Bashir x) tel(z(x1 x2 x3 x4 x5  y)))
```

Answer
```
(Khan EC1)
```

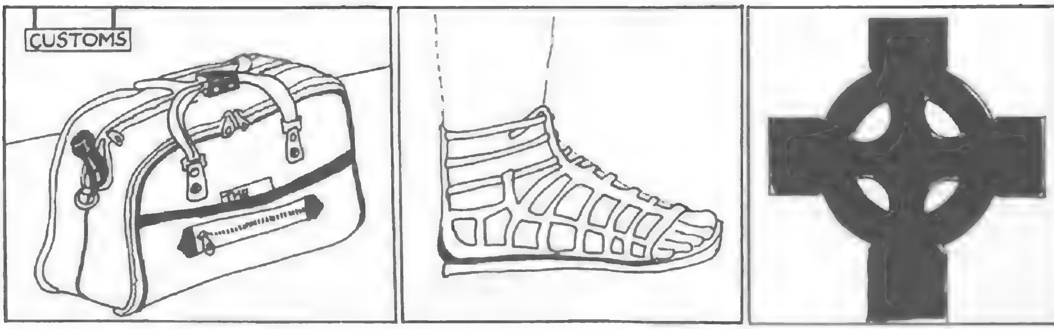
In the last question, variables were used to find the surname (x) and the postal area (y). z was used as a place holder for the telephone number, while x1, x2, x3, x4, and x5 were place holders for the words ´address 2a Bleeding Heart Yard´.

Activity 4
Make a database to list telephone/addresses. How many different kinds of question can you ask?

Activity 5
Try writing a database using this format:

```
(Xenia King) tel((12345)(address 45 Short
Hill)(postcode BD16 1AA)))
```

4 Animal, Vegetable or Mineral?



There is a guessing game that you may have played with your friends, which is called Animal, Vegetable or Mineral? One person thinks of an object or idea, while the others have to guess what it is. The only clue is whether it is animal, vegetable or mineral. You can use the same idea with interactive programs in PROLOG. To write an interactive program in which the computer can ask you questions, you use the PROLOG word **is-told**.

Loading TOLD
To use **is-told** you will need to load the program TOLD on the cassette. Type LOAD TOLD and press ENTER. You will find the program a little way after SIMTRACE and EXPTRAN.

Customs check
Some countries (in this program a particular Arab one) do not allow certain items through their customs. This highly simplified program is a check list which asks foreign visitors what they have in their luggage. If a restricted item is reported, the program asks the visitor to open his or her suitcase.

We will assume that customs officials have drawn up the following list of restricted items.

```
items not-allowed   Hi-Fi-equipment
items not-allowed   fire-arms
items not-allowed   Coca-Cola
items not-allowed   Elvis-Presley-records
items not-allowed   fruit
```

Coca Cola and Elvis records were in fact restricted items in Saudi Arabia during the 1960s.

To run this program you use a combination of **which** and **is-told**

```
which (x is a restricted item. Please open your
suitcase: items not-allowed x & (Do you have x)
is-told)
```

Let´s break this down into answer pattern and question pattern.

1. The answer pattern is

```
x is a restricted item. Please open your suitcase:
```

Remember you can write instructions or messages inside the answer pattern. These will be printed on the screen together with the answer to your question.

2. The question pattern is

```
items not-allowed x & ( Do you have x) is-told
```

Here we have two parts to the question pattern. The first is the pattern

```
items not-allowed x
```

This relates to the items listed in the database such as fruit or fire-arms etc. The second part is

```
&( Do you have x ) is-told
```

The question inside the brackets is again a ´free´ message. You can write anything you like. I could have written (Have you got x) or (Is there any x in your suitcase) and so on.

The word **is-told** makes the program ask you questions.

Copy in the data sentences and try questioning the data:

```
which (x:x is a restricted item.  Please open your
suitcase: items not-allowed x & ( Do you have x)
is-told)
```

Answer

```
Do you have fruit     No
Do you have fire-arms No
Do you have Coca-Cola Yes
Coca-Cola is a restricted item.  Please open your
suitcase
```

Airport security
This program checks for metals in your luggage. If you have something made of metal, you must open your suitcase.

```
suitcase contains scissors
suitcase contains suit
suitcase contains razor
suitcase contains socks
suitcase contains magazines
suitcase contains alarm-clock
```

```
which (x metal detected. Please open your
suitcase:suitcase contains x & (x made of metal)
is-told)
```

```
Answers
   suit made of metal?      NO
   razor made of metal?     YES
razor metal detected. Please open your suitcase
   socks made of metal?     NO
   magazines made of metal? NO
   penknife made of metal?  YES
penknife metal detected. Please open your suitcase
```

Roman dig
This program is another check list used for sorting ´things´ found on a Roman dig. The Chief Archaeologist wants anything made of mineral to be placed in a special box. Fred, a novice digger, uses PROLOG to check what he has found.

```
Fred finds corn-seeds
Fred finds sandal
Fred finds pottery
Fred finds spear
Fred finds rope
```

```
which (z is a mineral.  Place in Box 3:Fred finds z
&( z mineral) is-told)
```

```
Answers
   corn-seeds mineral?   NO
   sandal mineral?       NO
   pottery mineral?      YES
pottery is a mineral.  Place in Box 3
   spear mineral?        YES
spear is a mineral.  Place in Box 3
   rope mineral?         NO
```

I am not suggesting that you should use this program for a real dig. I am only outlining some of the uses of is-told as a check list in certain practical situations.

But what about a fantasy world?

Fraggle Rock

```
   Fraggles live-in rock
   Fraggles live-in light-house
   Fraggles live-in garden
   Fraggles live-in caves
```

```
   which( Let the music play down at Fraggle y:Fraggles
   live-in y &( Fraggles live in y) is-told)
```

```
Answers
   Fraggles live in rock?  YES
Let the music play down at Fraggle rock
   Fraggles live in lighthouse NO...
```

Recursion

An important feature of PROLOG is recursion. Recursion can be demonstrated by the following program

```
   Leeds part-of Yorkshire
   Bath  part-of Avon-county
   Yorkshire part-of England
   Avon-county part-of England
```

In this program we would like to define a new relationship **is-in**. For this we need two rules:

```
1.  x is-in y if x part-of y
2.  x is-in y if z part-of y and x is-in z
```

Let´s follow the computer as it searches the data and applies the two rules. Let´s start the program with the question

```
which(x:x is-in England)
```

The computer begins its search for the answer by going to the relation **is-in.** The first rule tells the computer to go and search the **part-of** relation. In the **part-of** relation, the data reads

```
Yorkshire part-of England
Avon-county part-of England
```

part-of is now the same as **is-in**, so that anything that is part of something is also in that something, according to the first rule.

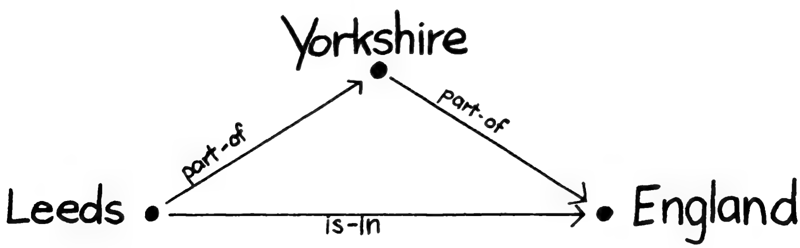
Yorkshire is now not only part-of England, but also in England. The same applies to Avon-county.

The computer now returns to the second rule. In this rule the computer is going to have to search the **part-of** and the **is-in** relations in order to return the required answers. This process that tells the computer to search the rule itself is called recursion. The rule that employs this process is called a recursive rule or definition. The computer goes to the second rule and finds a value for **x** but it also needs to find a value for **z** as well

```
z part-of y if x is-in z
```

If we look at **z** we can see that we are trying to find the area that is both part-of somewhere and has a place in it as well.

Look at the following network



and you will see that Yorkshire and Avon are both part of England and each has a town in it. Having found that **z** is in Yorkshire, the computer evaluates

```
z part-of y and x is-in z
```

or, in plain words, Yorkshire is a part of England and Leeds is in Yorkshire.

The computer knows that the first part is true from the data. It knows that the second part is true from the first rule and from the data.

What will it do next? The computer goes to the first part of the rule to see what it is that the rule has established. The first part of the rule states

```
x is-in y if...
```

The computer then sees that Leeds is in England and outputs the answer. This it does for each town in the database.

R Two rules

You will have noticed that two rules are used in this program:

```
(1)  x is-in y if x part-of y
(2)  x is-in y if z part-of y & x is-in z
```

The first rule is an ordinary rule, that is, it is not recursive. The second rule is recursive, since the relations **is-in** refers back to the relation being defined.

All recursive rules or definitions must be preceded by at least one ordinary or non-recursive rule or fact, otherwise they end up chasing their own tails. They become completely circular! Try running the program with the recursive rule first and see what happens. You should receive the error message - **"No space left"**.

Family tree

Tracing your ancestors or those of other people can be a fascinating and exciting experience. You can obtain information about family histories from church registers or town records. Some of these go back hundreds of years. You could be related to King Harold or William the Conqueror, or even Boadicea. The next program is another example of recursion, in which the relation ´ancestor-of´ is part of a recursive rule. Your ancestors are your parents and all the ancestors of your parents. To define ´ancestor-of´ we need this rule:

```
x ancestor-of y if z parent-of y and x ancestor-of z
(recursive rule)
```

Remember! A non-recursive rule is also needed, otherwise the program will be trapped in an endless loop. And remember that this non-recursive rule must be written first:

```
x ancestor-of y if x parent-of y
(non-recursive rule)
```

Putting these two rules together, we can define the relation ´ancestor-of´. For this example I have chosen to trace the family of Saint David, the patron saint of Wales.

```
(Vortigern parent-of Vortimer)
(Vortimer parent-of Anna)
(Anna parent-of Non)
(Gynyr parent-of Non)
(Non parent-of Saint-David)
(Sandde parent-of Saint-David)
(Edeyrn parent-of Cunedda)
(Gwawl parent-of Cunedda)
(Cunedda parent-of Ceredig)
(Ceredig parent-of Corwn)
(Corwn parent-of Sandde)
```

```
which (x:x ancestor-of Saint-David)
```

```
Answer
   Vortigern
   Vortimer
   Anna
   Gynyr
   Non
   Sandde
   Edeyrn
   Gwawl
   Cunedda
   Ceredig
   Corwn
   No (more) answers
```

The tree for Saint David (circa 562) traces the Saint´s ancestors back to King Vortigern of the Britons (circa 448) and to King Edeyrn of the Picts (circa 389). How far in recorded history can you trace your ancestry?

Activity 6
Can you complete this program?

Rules

```
(1) ?
(2) x joined-to y if ? connected-to x and ? joined-to
     ?
```

```
   (foot-bone connected-to ankle-bone)
   (ankle-bone connected-to leg-bone)
   (leg-bone connected-to knee-bone)
   (knee-bone connected-to thigh-bone)
```

```
   which (x:x joined-to y)
```

Answer
   ?

Lists and recursion
Recursion can also be used with lists, as the following example shows.

```
Rules
   x belongs-to (x|y)
   x belongs-to (y|z) if x belongs-to z
```

This means that **x** belongs to the list **(y|z)** if and only if **x** also belongs to the tail **z**, which is already a member of the list **(y|z)**. To check these rules you can ask the following.

```
Query
   which (x:x belongs-to (parrots swans))

Answer
   parrots
   swans
   No (more) answers

Query
   which(x:x belongs-to (football hockey))

Answer
   football
   hockey
   No (more) answers
```

The length of a list

Sometimes you might want to know how many answers there are to a particular question. For example you might want to know how many children there are in a family database. To do this you need to define a rule that will tell you the length of a list. This in turn uses another PROLOG word SUM.

SUM is a PROLOG word that can be used for addition and subtraction of whole numbers. Briefly (see Chapter 6 for a fuller explanation) it can be used to check a sum.

```
is(SUM 10 15 25)
```

Answer
```
YES
```

Or it can be used to add two numbers together. If you want to find the sum of 12 and 6, you can ask the following.

```
which(x:SUM(12 6 x))
```

Answer
```
18
No (more) answers
```

Let´s apply SUM to a rule that will find the length of a list. To do this you need two rules:

```
(1) () has-length 0
(2) (x|X) has-length z if X has-length y and SUM (y 1
    z)
```

The second rule looks rather complicated, but if we work through it step by step, we will see that it is not so difficult after all.

Supposing you have a data base with the names of the following children

```
(Pavlos Luke Suzanna)
```

these names can be represented by the variables **(x|X)**

```
(Pavlos Luke Suzanna () )
   x         X
```

When you are using heads and tails of a list, the list is written **(x|X)** in which the small **x** is the head of the list, that is Pavlos, and the big **X** is the tail,

that is Luke and Suzanna and the empty list (). The first rule tells the computer not to count the empty list - or rather to count it as zero. To calculate the length of a list, the computer will have to count 1 for the head and add it to the length of the tail. Let´s now look at the sentence after the ´if´ in the second rule

```
if X has-length y and SUM (y 1 z)
```

In this part of the rule we call the length of the tail **y** and go on to add to the tail **y** the number 1, which represents the head of the list - Pavlos. The result of this we call **z.** We can now go back to the conclusion

```
(x|X) has-length z
```

The answer is 3.

You can therefore ask the question

```
which(x:(Pavlos Luke Suzanna) has-length x)
```

and obtain the result 3. Of course this example only uses three names. In practice you would most likely be working with considerably larger numbers, within a very much larger database.

International athletics

This program uses (1) the rule to find the length of a list together with (2) the rule to find list membership. But first the data.

```
(Jones Davis Smith) represent UK
(Wien Taylor) represent USA
(Shinoda Nagana) represent Japan
(Wyatt Timms Stone White) represent NZ
```

The length of a list

```
() has-length 0
(x|X) has-length z if X has-length y and SUM (y 1 z)
```

The membership of a list

```
x belongs-to (x|y)
x belongs-to (y|z) if x belongs-to z
```

Combining these sets of rules, you can ask the following questions:

1. which(x:y represents USA and x belongs-to y)

Answer
```
Wien
Taylor  No (more) answers
```

This can be read as - who represents the USA? This outputs the list of names **(Wien Taylor)** as the answer, followed by the question

```
which x
```

that is, which names belong to that list of names. It then picks out the individuals by name. If on the other hand you ask

```
which(x:x represents USA)
```

the answer will be

```
(Wien Taylor)
```

The answer this time is the complete list of names, and not separate individuals as in the previous question.

2. Jones is part of a team, which one?

which(x:x represent y and Jones belongs-to x)

Answer
```
(Jones Davis Smith)
```
3. Which country does Jones represent?

```
which(y:x represent y and Jones belongs-to x)
```

Answer
```
UK
```
4. How many competitors represent UK?

```
which(x:y represent UK and y has-length x)
```

```
Answer
   3
   No (more) answers
```

This reads - if some athletes represent UK, how many are there?

5. How many athletes in each team?

```
   which(x:y represent z and y has-length x)
```

```
Answer
   3
   2
   2
   4
   No (more) answers
```

Better still, you could ask for the number of athletes in each team, together with the name of the country they represent:

```
   which(x z:y represent z and y has-length x)
```

```
Answer
   3 UK
   2 USA
   2 Japan
   4 NZ
 No (more) answers
```

Negation

Here is another database for you to consider

```
   Poole is-in England
   Norwich is-in England
   Tokyo is-in Japan
   Montreal is-in Canada
   Paris is-in France
   Bern is-in Switzerland
```

```
   x foreign-town if x is-in y and not x is-in England
```

The database is quite straightforward, but the rule uses a new term **not**. As well as writing positive rules which describe something, you can also use negative sentences. You can say what something is not. In this program, for example, Paris is a foreign town, since it

is in France and not in England. In order to use **not** in your rules, you must have a positive sentence before your negative sentence. You could not write for example

```
x foreign-town if not x is-in England
```

but on the contrary you must use a positive sentence after ´if´, that is **x is-in y,** followed by **not x is-in** England.

Let´s take another family tree - The Gotobeds (1715-1850).

```
Jeremiah parent-of Joshua
Jeremiah parent-of Martin
Joshua parent-of Elizabeth
Joshua parent-of Horace
Horace parent-of Egbert
Horace parent-of William
Elizabeth parent-of Louise
Jeremiah male
Joshua male
Martin male
Horace male
Egbert male
William male
```

```
x father-of y if x parent-of y and x male
x childless if x male and not x parent-of y
```

We would like to find out who are the men with no children.

```
which(x:x childless)
```

Answer
```
Martin
Egbert
William  No (more) answers
```

But what if we want to find out the names of all the members of the Gotobeds that are childless? The second rule only tells us about the male members of the

Gotobed family. If we want to find out all the members of the family who are childless, both male and female, we need to expand the data base. First we need to add some more facts.

```
Elizabeth female
Louise female
```

Having established who is male and who is female (the computer is quite dumb in these matters!), we need some more rules. And since we have to use a positive sentence before a negative sentence, we need to make up a positive sentence after ´if´. Let us use the relation ´person´.

```
x person if x male
x person if x female
x childless if x person and not x parent-of y
```

This new rule defining ´childless´ now includes both male and female members of the family.

We can now ask a question to find out all the members of the family who are childless.

```
which(x:x childless)
```

Answer

```
Martin
Egbert
William
Louise
No (more) answers
```

When you use **not**, the thing to remember is that you must precede the **not** or negative sentence with a positive sentence. If you forget, the program will not work.

If you wish to negate several sentences in your rules or questions, you must place the negated sentences inside brackets

```
which(x:x male and not(x father-of y and y male))
```

In this question, **not** is applied to the relations **father-of** and **male.** Type in this question and find out the answer for yourself.

Either...or

Sometimes you can define a relation using only one rule, which can however provide you with an alternative conclusion. This can be achieved by using **either...or.**

We could define parent-of using two rules as follows

```
x parent-of y if x father-of y
x parent-of y if x mother-of y
```

We can however redefine **parent-of** by using only one rule

```
x parent-of y if(either x father-of y or x mother-of y)
```

Using not and belongs-to

If we return to the rules for list membership, we can use these rules to check that something is not in a list:

```
x belongs-to (x|y)
x belongs-to (y|z) if x belongs-to z
```

```
which(x:x belongs-to (salt pepper mustard pickles
lettuce eggs) and not x belongs-to (lettuce eggs))
```

Answer

```
(salt pepper mustard pickles)
No (more) answers
```

Whether you might want to rush to the computer before filling your sandwiches is a matter of choice, but the idea of **not** with **belongs-to** can be applied to more complex programs. This particular example I have shown you here is merely to demonstrate how it works. As for more complex programs? It´s up to you!

Activity 7

Pogo-stick championships.

```
Jackson represents UK
Brown represents UK
Pinter represents USA
Gillespie represents USA
Keizo represents Japan
```

Rules
   x foreign-pogoist if x represents y and not x represents UK
   x British-pogoist if x represents UK

Can you find out who represents which country? Who is a foreign pogoist? Can you add to the data base so that you can ask - what on earth is a pogoist?

Building lists

Not only can you use lists to store information in your database, you can also program the computer to build lists for you from within the program itself. In certain programs, for example, you may wish the computer to output an answer in the form of a list.

```
Bill friend-of Ted
Sue friend-of Ted
Liz friend-of Ted
Cy friend-of Ted
```

How many friends has Ted? The data for this program is stored in the binary form, Ind - Rel - Ind. But you want the computer to pick out Ted´s friends and put them together in a list, count the list and give you the answer. To do this you need the PROLOG word **isall.** This word takes data sentences and puts them together in the form of a list. **isall** here is used in a question. The computer searches the database, picks out individual items of information and outputs them as a list. In this program **isall** is used with the rule **has-length.** The computer searches the sentences **x** friend-of Ted, picks each name, places them in a list called **y** and then counts them to give the answer - 4.

```
which(x:y isall(z:z friend-of Ted) & y has-length x)
```

In this question you want to find out how many friends **x** there are in the list **y**, which in turn asks who these friends **z** are. You then ask for a count of the names in the list **y** to give you the answer **x.** In other words you make a list **y** out of all the **z** who are friends of Ted and then you find the length of this list **y. isall** employs a question within the main question, that is

```
isall(z:z friend-of Ted)
```

You can also ask - who has four friends?

```
which(x:y isall(z:z friend-of x) & y has-length 4)
```

```
Answer
   Ted
   No (more) answers
```

APPEND
Another useful in-built facility of PROLOG is the word **APPEND.** This sticks or glues two lists together. Let´s have a look at an example.

```
(Doc-S Doc-H) work ENT-Department
(Doc-F Doc-C) work Casualty
```

From this data we want to output all the names of the Doctors in one list.

```
x employed-at General-Hospital if z work
ENT-Department and y work Casualty and APPEND(z y x)
```

```
all(x:x employed at General-Hospital)
```

```
Answer
   (Doc-S Doc-H Doc-F Doc-C)
   No (more) answers
```

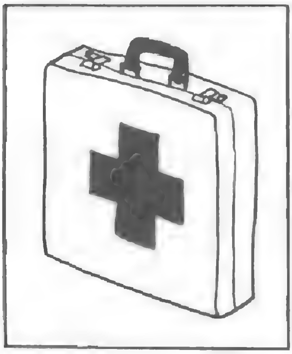
Looking at the last part of the rule we have

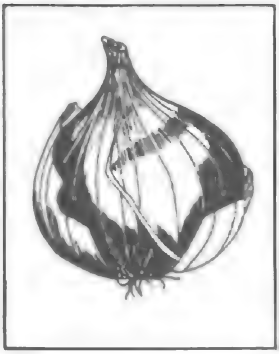
```
APPEND(z y x)
```

This means append, glue or stick the list z to the list y and output the answer as list x.

Some PROLOG tools
So far you have met rules to find the membership of a list, rules to find the length of a list and a command word (**isall**) that collects information and outputs it for you as a list. All these are useful tools for writing active databases in PROLOG. In the next chapter you will see how these tools may be applied to what are known as ´expert systems´.

5 Expert Systems





In Clarke and Kubrick's 2001: A Space Odyssey, there is a sinister computer called HAL. HAL is an expert system that not only pilots the Jupiter probe but also supports all the life systems aboard the spaceship. HAL appears to know everything, but if you know the story, he has a serious bug in his system - that eventually destroys the mission. Not all expert systems need be of the rogue variety and some in fact are useful tools, especially in such areas as medical research.

But what is an expert system? Researchers in the science of Artificial Intelligence have developed computer programs that try to solve problems using specialist, human knowledge. The computer is supplied with expert information by a specialist, for example a doctor or a geologist, who in turn uses the computer to process his knowledge. Instead of storing all your expertise in your head, you can store your knowledge in an expert system, which in turn can help you to process your knowledge. There is a program called MYCIN which attempts to diagnose infections in the blood. The expert knowledge is supplied by a specialist in medicine, who then uses the computer to carry out a diagnosis and to prescribe an appropriate course of antibiotics. The computer itself cannot tell the difference between a pint of blood and a bottle of milk, it is the expert knowledge fed into its memory that enables the computer to diagnose and recommend a cure. There is another expert system called PROSPECTOR which is based on expert knowledge supplied by geologists, and which tries to identify probable sites of valuable minerals. In fact PROSPECTOR turned out to be a winner. Geclogists using this expert system took the advice given by the computer to dig in such and such a place, and struck lucky!

Technically speaking, an expert system is a program

which is used to solve problems which would otherwise require specialist human knowledge or skills. An expert system could be used to bake a cake or for fault finding in electrical circuits. It all depends on your own expert knowledge, whatever that may be. Using a mini expert system on your Spectrum may not compare with MYCIN or PROSPECTOR, but it can give you a glimpse of the powerful ideas behind what is known as Artificial Intelligence or Knowledge Engineering.

Mini expert systems

The following programs are ideas for you to develcp, or to adapt to your own interests, hobbies etc. They are not ready-made programs. The last thing I want to do in this book is to dictate what should or should not be done with PROLOG. The field for creative ideas is there for you to explore - I hope I have given you the means to do this.

The following programs use the PROLOG word ´is-told´ in a rule. This sets up an interactive program in which the computer asks you questions.

First Aid

```
Toothache   remedy  (see Dentist)
Earache     remedy  (hot water bottle)
Headache    remedy  (take an aspirin)
Sore-foot   remedy  (rest it)
```

```
x complaint if x remedy y & (is x complaint) is-told
```

If we substitute Toothache for **x** in the rule, we get

Toothache complaint if Toothache remedy (see Dentist) and (is Toothache complaint)

The **is-told** asks the question - **(is Toothache complaint?).**

To start the program you type

```
which((x y):x complaint & x remedy y)
```

Answer

```
is Toothache complaint? yes
Toothache (see Dentist)
is Earache complaint? yes
Earache (hot water bottle)...
```

Weight-watcher

Whether you can resist eating the casserole served up before you or not, you may like to know whether it contains a high calory content, and if so, which ingredients to avoid.

```
(lean-chicken has low-cals) ingredient
(gravy has high-cals) ingredient
(tomatoes has low-cals) ingredient
(potatoes has high-cals) ingredient
(celery has low-cals) ingredient
(dumplings has high-cals) ingredient
(spices has low-cals) ingredient
(bacon has high-cals) ingredient
(rice has high-cals) ingredient
(prawns has low-cals) ingredient
(garlic has low-cals) ingredient
(onions has low-cals) ingredient
(noodles has high-cals) ingredient
(olive-oil has high-cals) ingredient
(octopus has low-cals) ingredient
```

```
(x|y) part-of casserole if (x|y) ingredient and
(Does your casserole contain x) is-told
```

(X/Y) is used here in order to output only the head of the list ´X´. We do not want all the words in the list. We only want ´lean-chicken´, ´gravy´, ´tomatoes´ etc. Try using X instead of (X/Y) and you will see what I mean!

```
X part-of casserole if X ingredient...
```

Remember we only want the head ´X´, not the tail ´Y´!

As with all rules in PROLOG, it is good practice to substitute words for the variables, in order to test the sense of what you are trying to say. To start this program you ask the following.

```
which(x:x part-of casserole)
```

You can if you prefer change **which** to **all** - it means the same thing:

```
all(x:x part-of casserole)
```

This question sets up an interactive program using **is-told**. The computer outputs the following:

```
Does your casserole contain lean-chicken? yes
(lean-chicken has low-cals)
Does your casserole contain gravy? no
Does your casserole contain tomatoes? yes
(tomatoes has low-cals)...
```

The computer will keep asking you questions until it has scanned all the database. If you answer ´yes´ to any of the ingredients in your casserole, the computer tells you whether they have a high or low calory content. If you answer ´no´, the computer goes on to the next question.

Spanish soups

There is a little cafe in Barcelona that serves two excellent soups. Here are the recipes

Catalonian Soup (Cat a l´onion, which contains onions and octopussy)
Recipe - spices, onions, tomatoes, noodles, olive-oil and octopus.
Costa Muchas (the same price as Catalonian)
Recipe - rice, celery, prawns, chicken, and garlic.

To question the data base about these mouth-watering delights, you need two rules:

```
(x|y) part-of Catalonian-soup if (x|y) ingredient
and (Does your soup contain x) is-told
(x|y) part-of Costa-Muchas if (x|y) ingredient and
(Does your soup have x) is-told
```

```
which(x:x part-of Catalonian-soup)
which(x:x part-of Costa-Muchas)
```

Answer

```
?
?
```

Holiday Guide

You might like to support your home town or area by writing a Holiday Guide which provides information on where to stay and what to see.

Before you compile your data, it is a good idea to ask some questions first. Questions could be:

1. Which towns have museums and sports facilities and provide bed and breakfast?
2. Which town near X has self-catering appartment to let?
3. Which bus goes to Y museum?
4. Are there any Youth Hostels?
5. What different places provide sports facilities and are beauty spots?
6. Can I go by bus to a historical place?
7. How can I get from Z to Y?

A holiday in West Yorkshire
Here is the beginning of a mini expert system to promote tourism in the Bradford Area.

Where to go and what to see:

```
((Ilkley Moor) (Baildon Moor) (Shipley Glen) (White
Wells) (Bolton Priory) ) beauty-spot
((Ilkley Moor)  (Baildon)) sports-facilities
Baildon sports (hang-gliding golf model-aeroplanes
walking)
Ilkley sports (riding rock-climbing)
Shipley sports (swimming squash)
Bradford sports (ice-skating swimming)
Bradford museums ( (Cartwright Hall) (National
Photographic) (Bolling Hall) (Industrial Museum))
Ilkley museums (White Wells)
Keighley museums (Cliffe Castle)
(Bolling Hall) location ((off the A650 Wakefield Rd)
(Bus 621 from Bradford))
(Cartwright Hall) location ((Lister Park) (on A650
Keighley Rd) (Bus 623 from Bradford or Manor Row))
(Cliffe Castle Keighley) location ((off A629 Skipton
Rd) (20 mins walk from railway station))
(Industrial Museum) location ((off A658 Harrogate Rd)
(Buses from Bank St))
((Bolton Priory) (White Wells) (Bolling Hall))
historical-site
(Bradford Ilkley Keighley) theatres
```

Where to stay

```
(Bradford Haworth Ilkley Keighley) hotels
(Bradford Ilkley Keighley) b-and-b
(Haworth Ilkley) self-catering
Bradford hotel-list((Unicorn Hotel) (Red Lion)
(Norfolk Gardens))
Ilkley hotel-list ((Royal) (Crown))
Haworth hotel-list ((Black Bull) (Old White Lion))
Keighley hotel-list ((Kings Head) (Dalesman))
Bradford b-and-b ((Contact Tourist info))
Ilkley b-and-b (Pondell Hall)
Keighley b-and-b (Fairleigh Guest House)
Haworth self-let ((Lothersdale, chalet in Bronte
country, tel 0535-32533) (Haworth Cotts,
tel 0532-752977))
Ilkley self-let (Burnsall Cotts,
tel 075672-668)
Youth-Hostels (contact YHA regional office, Bingley
595)
```

How to get there

```
(Bus 621) connects (Bradford Bowling-Hall-Rd)
(Bus 623) connects (Bradford Manor-Row Lister-Park
Shipley)
(Bus 625) connects (Bradford Manningham Shipley)
(Bus 627) connects (Bradford Bank-St Moorside-Rd)
(Train BI) connects (Bradford Ilkley)
(Train BC) connects (Bradford Keighley Carlisle)
```

R

I have used the rules for list membership and for list length together with **isall**(How many buses go to Shipley?)

```
x belongs-to (x|y)
x belongs-to (y|z) if x belongs-to x

() has-length 0
(x|X) has-length z if X has-length y and SUM (y 1 z)
```

?

1. How many museums are there in Bradford?

```
which(x:Bradford museums y and y has-length)
```

Answer

```
4
No (more) answers
```

2. How many buses go to Shipley?

```
which (x:y isall(z:z connects Y and Shipley
belongs-to Y) and y has-length x)
```

Answer

```
2
No (more) answers
```

3. Are there any Youth Hostels?

```
is(Youth-Hostels x)
```

Answer

```
YES
```

4. How do you get from Manningham to Shipley?

```
which(x:x connects y and Manningham belongs-to y)
and Shipley belongs-to y)
```

Answer

```
(Bus 625)
No (more) answers
```

5. Which is both a beauty spot and a historical site?

```
which(x:y historical-site and z beauty-spot and x
belongs-to y and x belongs-to z)
```

Answer

```
(Bolton Priory)
(White Wells)
No (more) answers
```

6. Which towns have museums, sports and bed/breakfast?

```
which(x:x museums y and x sports z and X b-and-b
and x belongs-to X)
```

Answer

```
Bradford
Ilkley
No (more) answers
```

Personal-Doctor

This program comes in two modes. The first mode - let´s call it the ´query mode´, is a normal database, which you can question like all the other programs in this book. The second mode, the ´advice mode´, is a special program whereby the computer asks you a question and then gives you some advice.

The computer asks - ´Is there something wrong with you?´ You can answer by typing in the name of your suspected complaint. For example, you might have a headache, aching limbs and a fever. You suspect you have a bout of influenza, so you type - ´ans flu´. The computer searches through the database until it finds a match for the word ´flu´. It then outputs the answer

(Seek advice from your doctor or obtain tablets from your local chemist)

This program is a medical database, but you could, if you wished, adapt this type of program to any other subject. It could be equally used for car maintenance or for testing subaqua equipment for example.

The second mode, which I have called the ´advice mode´, introduces you to a new and powerful idea in expert systems - that of the computer asking questions. At this level of programming, the computer is still a dumb machine but, with a little bit of imagination and programming skill, you could create an almost ´intelligent´ database. You might even make your friends think that your computer really has a mind of its own!

In larger expert systems of the MYCIN or PROSPECTOR type, the computer asks for specific information from the human expert, to help it to produce the final result. Some of these sophisticated machines can carry out very high powered conversations.

Personal-Doctor is not an ´intelligent´ system like MYCIN, it only pretends to understand your problem, but it is a useful tool all the same, for processing large amounts of data and providing you with a quick and logical answer.

Mode 1

```
measles has-symptoms (rash fever cough)
flu has-symptoms (cough fever catarrh headache
aching-limbs)
dreaded-lurgy has-symptoms (fever aching-knee-caps)
eye-strain has-symptoms (headache watery-eyes)
migraine has-symptoms (headache eye-strain)
food-poisoning has-symptoms (rash fever tummy-pain)
shingles has-symptoms (rash fever)
```

R

```
x belongs-to (x|y)
x belongs-to (y|z) is x belongs-to z
```

I have a fever - what could be wrong with me? This is the same as asking - which illnesses have a fever as a symptom?

```
which(x:x has symptoms y and fever belongs-to y)
```

Answer
  ?

Mode 2

```
flu has-remedy (Seek advice from your doctor or
obtain tablets from your local chemist)
measles has-remedy (rest from all activities and
smother yourself with horse linament)
eye-strain has-remedy (refrain from reading and
watching TV or obtain glasses from your local
optician)
food-poisoning has-remedy (don´t eat any solids and
drink only fluids such as black-tea or boiled water
and seek advice from your doctor)
```

You could add a safety clause:

```
x has-remedy (seek advice from your doctor)
```

For any complaint you should seek expert medical advice and not just swallow tablets or pills.

```
y question if (Is y wrong with you) is-told
```

```
all((x|z):y question and y has-remedy (x|z))
```

Note to hypochondriacs
If you answer only yes to the computer´s question - ´Is x wrong with you?´, the computer will print out all the remedies - you can take your pick!

Bake-a-cake
If you are a successful cook you will not need this rather modest, mini - expert system. You will be able to aspire to higher things, using a PROLOG expert system to diagnose faults in more advanced recipes or

to plan a special banquet. But for those of you who are baking your first cake, this program might be useful to get you started.

The success in producing a perfect cake depends on several factors, which I have listed in the database entitled 'Cake problems'. If things go wrong you can question the computer. However before you bake your cake, you will need to know what basic ingredients to use. A separate database called 'Ingredients' has been provided for this purpose.

Ingredients

```
(8 oz margarine or butter) ingredients
(8 oz sugar) ingredients
(8 oz or 4 eggs) ingredients
(8 oz plain flour) ingredients
(1.5 teaspoons of baking powder) ingredients
oven temperature moderate
(45 minutes) cooking time
```

```
1. Which ingredients do I need?
   which(x:x ingredients)
2. At what temperature?
   which(y:oven temperature y)
3. How long in the oven?
   which (Y:Y cooking time)
```

Cake problems

```
(soggy texture) means (insufficient eggs in your
cake)
(solid lump) means (no baking powder)
(mountain-shape with cracked and burnt surface)
means (oven is too hot)
(sunken in the middle) means (cake disturbed by
opening oven door too soon)
```

```
z question if (Is z a problem with your cake)
is-told
```

What's wrong with my cake!?

```
all((x|y):z question and z means (x|y))
```

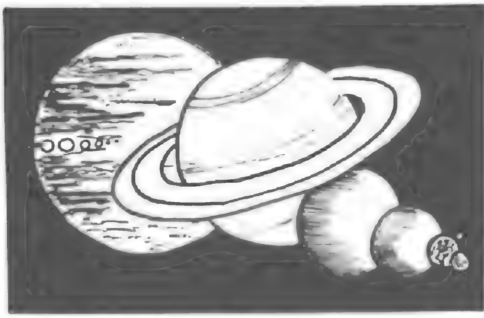
The computer responds by printing

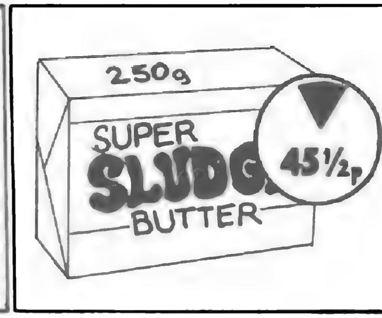
```
Is X a problem with your cake?  ans (soggy texture)
(insufficient eggs in your cake)
Is X a problem with your cake?...
```

Your answer must be a list, that is you must place your answer in brackets to match the information stored in your database.

See if you can develop this expert system to cover other areas of baking and cooking in general. Perhaps like that famous Saxon king, Alfred the Great, you will discover other problems as you improve your cooking.

6 PROLOG Arithmetic





Addition and subtraction (difference)
You have already met the PROLOG word **SUM**, in the rule to find the length of a list:

```
SUM(y 1 z)
```

Missing numbers
To add two numbers together, you ask a question to find the missing number. The missing number ´x´ is in fact the answer to your SUM.

```
which(x:SUM(30 43 x))
```

Answer
   73

The same can be applied to the addition of decimals.

```
which(x:SUM(6.6 4.31 x))
```

Answer
   10.91

To subtract (find the difference between) two numbers you again ask a question, but the variable ´x´ is placed before the answer. This time you have been given the answer. What you have to do now is find the missing number to complete the SUM.

```
which(x:SUM(x 23 89))
```

Answer
66

```
which(x:SUM(34 x 157)
```

Answer
123

Multiplication and division

You can also use PROLOG to multiply and divide numbers. To do this you use the PROLOG word, **TIMES**.

Once again, to carry out the operation, you ask a question.

```
which(z:TIMES(7 8 z))
```

Answer
56

```
which(y:TIMES(32.7 51.23 y))
```

Answer
1675.221

To divide a number, you use **TIMES** again but place the variable (x y z etc..) before the answer.

```
which(x:TIMES(x 6 48))
```

Answer
8

```
which(y:TIMES(8 y 48))
```

Answer
6

Limitations

When you use **SUM** or **TIMES**, you are limited to two numbers only. If you try to calculate with more than two numbers you will receive the error message ´Too many variables´. However, you can combine operations in one single number sentence.

Try this one:

```
which(x:TIMES(45 23 y) & SUM(y 73 x))
```

The answer **y** to the first operation (TIMES) is passed on to the variable **y** of the second operation (SUM), which outputs the answer **x**.

Answer
   1108

Using extra words
The variable box may be used to give more detailed answers to arithmetical questions. For example, in division you may wish to have the word ´remainder´

printed before the figure in your answer.

```
which((x remainder y):TIMES(4 x 23 y))
```

Answer
   (5 remainder 3)

```
which((z remainder X):TIMES(z 3 19 X))
```

Answer
   (6 remainder 1)

Checking arithmetical operations
You can check your arithmetic by using **is**.

```
is(SUM(25 25 100))
```

Answer
   YES

```
is(TIMES(7 7 49))
```

Answer
   YES

Sometimes you may wish to check whether the result of division is an exact result, that is, whether a number is exactly divisible by another.

```
is(TIMES(4 y 36) and y INT)
```

```
Answer
   YES
```

We have a new term here, **INT.** This word checks that numbers are whole numbers or integers and outputs in this case the answer - YES.

```
   is(TIMES(x 5 71) and x INT)

Answer
   NO
```

Using LESS

The PROLOG word **LESS** can be used to carry out arithmetical operations within your rules, as the following program demonstrates.

Solar System

```
  Mercury has-diameter 3008 (miles)
   Venus has-diameter 7600
   Earth has-diameter 7927
   Mars has-diameter 4200
   Jupiter has-diameter 88439
   Saturn has-diameter 75060
   Uranus has-diameter 30875
   Neptune has-diameter 33000
   Pluto has-diameter 3600
   The-Moon has-diameter 2160
   Purple-Planet has-diameter 1000
```

The Sun, which is the centre of our Solar System, has a diameter of approximately 865,000 miles. The Earth is just one of nine (or ten!) planets that revolve round the Sun.

```
   x larger-than y if x has-diameter z and y
   has-diameter X and X LESS z
```

This reads, for example
The-Moon is larger than the Purple-Planet if The-Moon has a diameter of 2160 miles and the Purple-Planet has a diameter of 1000 miles and 1000 is LESS than 2160 - which it is!

```
is(The-Moon larger-than Purple-Planet)
```

Answer
```
YES
```

Let's ask some more questions about our Solar System.

1. Which planets are larger than Neptune?

```
which(x:x larger-than Neptune)
```

Answer
```
Jupiter
Saturn
No (more) answers
```

2. Which planets are smaller than Neptune? Remember that whereas you can understand what this question is asking, the computer cannot. It only knows the relation 'larger-than'. Therefore to ask this question you have to juggle with the idea of 'smaller than'. You already know that Jupiter and Saturn are larger than Neptune; therefore all the other planets must be smaller. But how do you ask the database? (Hint - you could use negation).

3. This question not only asks which planets are larger than the Purple Planet, but what are their respective diameters?

```
which((x y):x larger-than Purple-Planet and x
has-diameter y)
```

Answer
```
(Mercury  3008)
(Venus    7600)
(Earth    7927)
(Mars     4200)...
```

What are the missing planets?

Let´s add two new rules to this program

```
(1) x lesseq x
(2) x lesseq y if x LESS y
```

We use these two rules to define the relationship ´less than or equal´ (**lesseq**). You can now find which planet(s) are the same size or smaller than the others.

4. Which planets are the same size or smaller than Pluto?

```
which(z:Pluto has-diameter y & z has-diameter x & x
lesseq y)
```

Answer

```
The-Moon
Purple-Planet
```

In this question, the computer uses **LESS** in the two new rules (**lesseq**), to calculate which diameters are equal to or smaller than the diameter of Pluto. There is plenty of scope here to develop this program further. You could write a rule to define ´greater-than´. Expand, adapt and experiment with this program and others and see what you can do with them.

Great Rivers

The same rule as for Solar System is used in the next program.

```
Nile has-length 4160
Amazon has-length 4050
Mississippi-Missouri has-length 3710
Yangtze has-length 3400
Danube has-length 1770
```

```
x longer-than y if x has-length z and y has-length X
and X LESS z
```

1. Which river is the longest?
2. Which river is longer than the Amazon?

```
which(x:x longer-than y)
```

Answer
```
?
```

```
which(x:x longer-than Amazon)
```

Answer
```
Nile
```

Activity 8
The World's largest cities
Use the following database to find out about the populations of cities throughout the world. Write your own rules to output the comparative size of cities together with their population sizes. Of course, add to the database if you wish.

```
Shanghai  pop 11000000
Mexico-City pop 9618346
New-York pop 7895563
London pop 6918000
Santiago pop 3724450
Rome pop 2914640
Manchester pop 2663500
Paris pop 2299830
```

Other uses of LESS
Like **SUM** and **TIMES**, **LESS** can be used to check arithmetical statements. However it only works with integers (whole numbers). You cannot use it to check fractions or decimals.

```
is(3 LESS 18)
```

Answer
```
YES
```

```
is(18 LESS 4)
```

Answer
```
NO
```

LESS with words
Not only can **LESS** be used to check numbers, but it can also be used with words. In this respect **LESS** operates according to alphabetical order.

```
   is(eggs LESS elephant)
```

Answer
   YES

Both words begin with ´e´, but the ´g´ in egg comes before the ´l´ in elephant in the dictionary or alphabet.

```
   is(Aubergine LESS Carrot)
```

Answer
   YES

```
   is(Leeds LESS Bristol)
```

Answer
   NO

Great Inventions and Discoveries
This program includes some inventions and discoveries, dating back from the fifteenth century, which have made a considerable impact on our society.

```
(aeroplane)date ((Wright Brothers) (USA 1903))
(Clock, pendulum)date ((Huygens) (Netherlands 1656))
(Bicycle)date ((Macmillan) (Britain 1839))
(Lift)date ((Otis) (USA 1852))
(Lightning conductor)date ((Franklin) (USA 1752))
(Printing)date ((Gutenberg) (Germany 1440))
(Balloon)date ((Montgolfier Brothers) (France 1783))
(Penicillin) date ((Fleming) (Britain 1929))
(Radium)date ((Curie) (France 1898))
(X-rays)date ((Roentgen) (Germany 1895))
```

We can easily question the database to find out who invented or discovered what. But suppose we want to know in which century a particular invention or discovery occurred. To answer this kind of question, we need to write a series of rules that will define the word ´century´ for each specific century.

R

```
x century twentieth if x date y and 1900 lesseq y
and y LESS 2000
x century nineteenth if x date y and 1800 lesseq y
and y LESS 1900
x century eighteenth if x date y and 1700 lesseq y
and y LESS 1800
x century seventeenth if x date y and 1600 lesseq y
and y LESS 1700
x century sixteenth if x date y and 1500 lesseq y
and y LESS 1600
x century fifteenth if x date y and ? lesseq y and y
LESS ?
```

Can you complete the missing figures?

Assuming that you still have the definition of **lesseq**, in your computer, you can proceed to question the database.

?

```
which(x:x century seventeenth)
```

Answer
(Clock, pendulum)

```
which(x:x century eighteenth)
```

Answer
(Lightning conductor)
(Balloon)

Supermarket

The PROLOG calculator can be used to work out the price of your shopping at the supermarket.

Items listed in the shopping list are pre-packed in Polythene and are priced in pence.

Shopping List

```
Apples cost 55
Bread cost 45
Cucumber cost 39
Lettuce cost 22
Potatoes cost 11
```

Sue goes to the supermarket and buys one packet of apples and two cucumbers. How much does she spend?

To find Sue´s total expenditure **x**, you can ask

```
which(x:Apples cost y and Cucumber cost z and
TIMES(y 1 X) and TIMES(z 2 Y) and SUM(X Y x))
```

Now this is not a very efficient way for Sue to add up her shopping list. To make the task easier, she needs another PROLOG facility which uses arithmetical expressions. To obtain this facility you will need to LOAD EXPTRAN. It can be found on the cassette, just after SIMTRACE.

Having loaded EXPTRAN, you can now use the five expressions, **+**, **-**, *****, **/** **and** **=**. (***** is the computer symbol for multiply, while / is the symbol for divide).

With the EXPTRAN facility, Sue can now ask

```
which(x:Apples cost y & Cucumber cost z & x =
(y*1+z*2))
```

For simple calculations, you can dispense with SUM and TIMES and simply use **which** in the following way.

```
which(x:x =(2+11+10))
```

Or as follows

```
which(y:y =(3*5+9))
```

You can even dispense with **which** and use the symbol **#**:

```
#(3*5+9)
```

Here is a mixed assortment for you to experiment with:

```
which(z:z =(30*60/3+2))
   #(100*100 -2/4)
which(x:y =(55*7)& z=(79+2) & SUM(y z x))
which(x:X =(69+4 & z=(92*5) & x =(X+z))
   #(2312/2*6 -3)
```

Note

It is advisable to use the symbol ´~´ for subtraction, since unless you are very careful in your use of spaces, the computer will interpret the subtraction symbol as a normal hyphen.

Appendix 1: Error Messages, Hints and Tips

Error messages
As you write and apply your programs you will inevitably come across error messages. These vary but some of the more common ones are as follows.

Number errors

3 Control error. You are using too many variables, or not enough, or you are using the wrong value for a variable. Also applies to an invalid form of sentence.

4 ADDCL error. You are trying to use a relation word that is a PROLOG word (**add is which**...), or it is the name of a currently opened file.

5 File error. This happpens if you try to use a file name that is also the name of a relation word, or if you try to load a file that is already loaded.

6 Too many files error. You are trying to open or load too many files at the same time. This error number also appears if you have trouble loading a program. Solution - reload your program.

Message errors
Out of space. That´s what it means!

File not found. You did not load your program correctly.

Syntax error. You used incorrect word order, or you used too many closing brackets. (If you are not sure how many closing brackets to use, wait for the computer to prompt you.)

Snippets
In this section I would like to conclude with a few snippets that you will find useful when programming.

Your own commands
You can write your own command words in the place of **is** or **all(which)**.
To do this type

```
x find if x all
```

or

```
x check if x is
```

Try asking questions with these new words:

```
find(x:x lives-in city)
```

or

```
check(x lives-on surface)
```

You can use any word you like as the following example shows:

```
x who-on-earth if x all

x do-tell-me if x is
```

```
who-on-earth(x:x lives-on Purple-Planet)
do-tell-me(x lives-on Purple-Planet)
```

Checking your database
You can also find the names of your relation words by typing

```
all(x:x dict)
```

or

```
list dict
```

This will output on the screen all the relation words in the database.

Sometimes a relation word may not belong to a current sentence in the database. For instance you may have entered the relation word **belong-to** and then subsequently deleted the sentences that use this word. The relation word **belong-to** still remains in the

database. To find out if it is still contained in a sentence you can ask

```
is(R defined)
```

where **R** stands for the relation word you are seeking.

You can combine these operations by asking whether a certain relation word is in the database, and whether or not it is still defined.

```
all(x:x dict and x defined)
```

Printing on the screen

To print ordinary text on the screen PROLOG uses two in-built functions **P** and **PP**.

PP ("a sausage") will print

```
"a sausage"
```

in full with quotes.

P ("a sausage") will print

```
a sausage
```

without the quotes. Words to be printed must be placed inside brackets. You are in fact asking the computer to print a list.

Appendix 2: Standard Syntax

Standard Syntax
No book on Micro-PROLOG would be complete without looking at Micro-PROLOG itself. Throughout this book you have been introduced to some of the important features of Micro-PROLOG through a more user-friendly syntax called SIMPLE.

Whenever you use SIMPLE to write your programs, the computer translates your data into what is called Standard Syntax. The computer accordingly translates back from Standard Syntax into SIMPLE whenever you ask a question. The computer always stores SIMPLE programs in Standard Syntax. This is the language directly understood by Micro-PROLOG.

This means that a sentence entered using SIMPLE is automatically stored in the computer as a sentence in Standard Syntax.

Using Standard Syntax
You can however by-pass SIMPLE and write your programs directly in Standard Syntax. Whether you have loaded SIMPLE or not, you have direct access to Standard Syntax. You can, if you like, first load SIMPLE and then use both forms of Micro-PROLOG. But before you do this, I would like to introduce you to yet another version of Micro-PROLOG called MICRO.

MICRO
If you load the other side of the cassette, that is, the opposite side of SIMPLE, you will find a program called MICRO.

This program uses both Standard Syntax and certain command words which are similar to SIMPLE. MICRO is a good introduction to Standard Syntax, and is therefore worth while using before you attempt Standard Syntax by itself.

To load this program, you type LOAD MICRO.

Sentences in MICRO
Sentences in MICRO are written directly in Standard Syntax. They are generally called clauses rather than

sentences. They are always written with the relation word first, and with the clause (sentence) enclosed by double brackets:

```
((lives-in Zorg city))
((found-on water surface))
```

Let´s compare some MICRO and SIMPLE clauses:

MICRO	**SIMPLE**
((found-in fuel city))	(fuel found-in city)
((dangerous Nurks))	(Nurks dangerous)
((visitor Capt-Tee))	(Capt-Tee visitor)

To add these clauses to the database you simply place them in double brackets and press ENTER. Unlike SIMPLE you do not need to use the command word **add.**

Asking questions

You can also ask questions using **is** and **which,** although you do not use a colon (:) with variables.

```
   which(x(lives-in x city))

Answer
   Zorg
   No (more) answers


   is(( found-in x city))

Answer
   YES


   which(z(dangerous z))

Answer
   Nurks
   No (more) answers


   which(Y(visitor Y))

Answer
   Capt-Tee
   No (more) answers
```

```
   is((robot Nurkl))
```

Answer
```
   NO
```

Here is some more data from the Purple Planet

```
   ((likes Zorg Selina))
   ((likes Selina Zorg))
   ((found-on food surface))


   is((likes Zorg Selina))
```

Answer
```
   YES


   is((likes Selina Zorg))
```

Answer
```
   YES


   which(x(likes x Selina))
```

Answer
```
   Zorg
   No (more) answers


   which((X Y) (found-on X Y))
```

Answer
```
   (food surface)
   No (more) answers
```

Rules in MICRO

Whenever you wish to define a relation, the rule is written with the relation word first; but with a difference - **if** is not used and neither is **and.** Thus

```
   ((friend x)(likes x y)(likes y x)
```

This means that x is a friend (if) x likes y (and) y likes x. The (if) and the (and) being omitted in the rule.

```
   which(x(friend x))
```

```
Answer
   Zorg
   Selina
   No (more) answers
```

More rules

```
1.  ((belongs-to x (x|y))
    ((belongs-to x (y|z) (belongs-to x z))
2.  ((has-length () 0))
    ((has-length (x|y)z) (has-length y X) (SUM X 1 z))
```

From these examples you can see that clauses in MICRO are virtually the same as sentences in SIMPLE. All you have to remember is to write the relation first and use double brackets. Remember too that **if** **...and** are omitted.

Try writing some of your SIMPLE programs in MICRO.

It´s the real thing

In this appendix I have only given you a glimpse of MICRO, just enough I hope to whet your appetite for more advanced forms of programming in PROLOG. To conclude I would like to look briefly at the ´real´ thing, that is Standard Syntax or Micro-PROLOG in its undiluted form.

Like in MICRO, clauses are entered using Standard Syntax:

```
   ((friend Zorg))
   ((friend Selina))
   ((friend Capt-Tee))
```

Although you do not need an add command to enter this data, there is an in-built command called **ADDCL** (ADD THE CLAUSE), which is used to place clauses with the same relationship at a specified position in the database.

Thus to place

```
   ((friend Astra))
```

as the first clause after the clause

```
   ((friend Zorg))
```

which we will assume is already the first clause in the program, you type

```
   ?((ADDCL((friend Astra))1))
```

Now type **LIST friend** and the following should appear:

```
((friend Zorg))
((friend Astra))
((friend Selina))
((friend Capt-Tee))
```

The use of **?** with **ADDCL** is a special command to carry out this operation. The figure 1 indicates where the clause is to be positioned.

Let´s type some more

```
?((ADDCL ((friend Doc))3))
?((ADDCL ((friend Zenith))2))
```

Type **LIST ALL**

```
((friend Zorg))
((friend Astra))
((friend Zenith))
((friend Doc))
((friend Selina))
((friend Capt-Tee))
```

If you wish to delete any of these clauses - for example **((friend Doc))**, you type

```
?((DELCL friend 3))
```

that is **DELETE CLAUSE friend 3.**

Asking questions

You can query the database by using the symbol **?**

If you want to know if anything is found on the surface of the Planet you can ask the following.

```
?((found-on x surface))
```

Answer

```
&.
```

If the answer is YES, the prompt sign **&.** appears. If the answer is NO another **?** symbol appears.

```
?((found-on robot surface))
```

Answer

```
?
```

If on the other hand you would like a printed result, then you must program the question yourself:

```
?((found-on x surface)(PP x))
```

This question uses the **PP** command. This is another in-built command which prints text on the screen. The answer to this question is
food.

It does not output the monsters (the Nurks...) or water. To do this we have to write our own definition of **which:**

```
((WHICH (X  Y))
(FORALL Y ((PP X)))
(PP That´s the lot!))
```

Now we can ask the question again.

```
WHICH(X(found-on X surface))
```

Answer

```
food
water
Chief-Nurk
Nurk1
Nurk2
Nurks
That´s the lot!
```

The definition of **WHICH**

In this rule
1. The **X** is the answer pattern.
2. The **Y** is a list of all the clauses that say **found-on surface.**
3. **FORALL** is another in-built PROLOG word that checks the list **Y** and finds all the clauses **found-on surface** that are in the database.
4. For each **Y** that has an answer, print (**PP**) the answer **X.**

Asking more questions

Data

```
((likes Astra visitor))
((likes Zenith visitor))
((likes Zorg visitor))
```

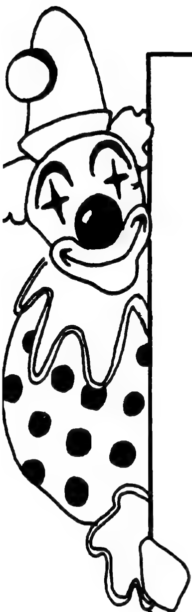
Query

```
?((likes x y)(PP x y))
```

```
Answer
   Astra visitor

Query
   WHICH(X(likes X visitor))

Answer
   Astra
   Zenith
   Zorg
   That´s the lot!

Query
   WHICH(Z(likes Astra Z))

Answer
   visitor
   That´s the lot!

Query
   WHICH(X(likes X Y))

Answer
   Astra
   Zenith
   Zorg
   That´s the lot!
```

And some more:

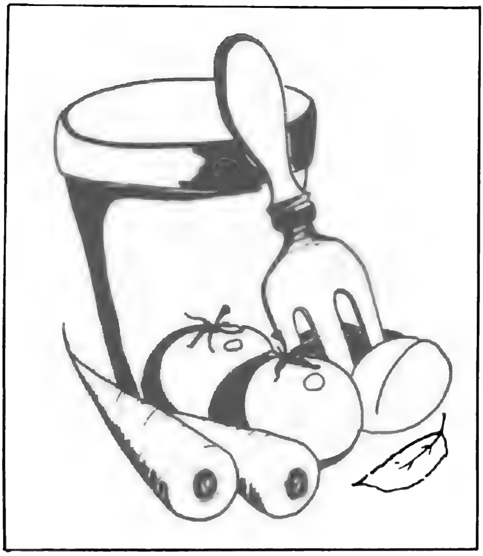
```
Data
   ((WHO LIVES IN THE CITY)(PP I DO))
   ((WHO ARE YOU)(PP ZORG THE ROBOT))

Query
   ?((WHO LIVES IN THE CITY))

Answer
   I DO

Query
   ?((WHO ARE YOU))

Answer
   ZORG THE ROBOT
```



Joke Box

```
Data
    ((KNOCK KNOCK WHO´S THERE)(PP AMOS))
    ((AMOS WHO)(PP A MOSQUITO))

Query
    ?((KNOCK KNOCK WHO´S THERE))

Answer
    AMOS

Query
    ?((AMOS WHO))

Answer
    A MOSQUITO
```

Appendix 3: Sample Programs



If you wish to grow your own food, you must know what and when to plant and how to recognize and control pests and diseases that are associated with fruit and vegetables. You will also have to consider other factors such as location, wind, temperature and seasonal change. All these factors can be built into a database using PROLOG.

The program begins with a gardener´s calendar, that tells you what to plant and sow and at what times. The data is classified according to a particular month, and the fruit, veg and general-work associated with that month. Gardeners using this program will need to modify some of the information to suit their particular area of the country.

Month by month

January

```
add((general-workl veg fruit) Jan)

add((Dig all available ground) general-workl)
   ((greenhouse jobs) general-workl)
   ((force rhubarb in the dark) general-workl)
   ((lift and blanch chicory) general-workl)
   ((protect cauliflower curds) general-workl)
   ((sow small salads under glass) general-workl)
   ((spray fruit trees with tar oil)
     general-workl)
   ((complete fruit tree pruning) general-workl)
```

```
add((order seeds and seed potatoes) vegl)
   ((sow tomatoes for June crop in greenhouse)
     vegl)
   ((sow leeks in greenhouse) vegl)
   ((sow French beans in greenhouse) vegl)
   ((sow shallots in open ground) vegl)

add((start early peaches, nectarines and
     apricots) fruitl)
```

February

```
add((general-work2 veg fruit) Feb)

add((complete digging & break down soil)
     general-work2)
   ((ventilate frames according to weather)
     general-work2)
   ((sow small salads for succession)
     general-work2)
   ((pollinate and disbud early fruit)
     general-work2)

add((sow broad beans under glass) veg2)
   ((sow lettuces, peas & cauliflowers in
     greenhouse) veg2)
   ((sow cucumber for early crop) veg2)
   ((plant Jerusalem artichokes) veg2)
   ((sow celery in frames) veg2)
   ((sow turnips, parsnips, spring-cabbages, in
     open ground) veg2)

add((prune Autumn-fruiting raspberries) fruit2)
   ((start strawberries under glass) fruit2)
   ((start fruit in unheated greenhouse) fruit2)
```

March

```
add((general-work3 veg fruit) Mar)

add((make successional sowings) general-work3)
   ((tend peaches etc) general-work3)

add((sow cauliflower, brocoli in frames) veg3)
   ((sow onions, spinach, broad beans and celery)
     veg3)
   ((plant rhubarb, horseradish & seakale) veg3)
   ((plant cucumbers) veg3)
   ((plant potatoes) veg3)

add((plant strawberries) fruit3)
   ((spray apple trees) fruit3)
   ((protect early fruit blossom) fruit3)
```

April

```
add((general-work4 veg fruit) Apr)

add((harden off onion, leek, lettuce in frames)
     general-work4)
   ((prick off seedlings) general-work4
   ((prepare celery trenches) general-work4)
   ((make successional sowings) general-work4)
   ((thin peaches and nectarines) general-work4)

add((nitrate of soda for spring cabbage) veg4)
   ((plant potatoes, onion, artichokes) veg4)
   ((sow kohl rabi, carrots, winter greens) veg4)

add((spray apples & pears) fruit4)
   ((lime sulphur for gooseberries) fruit4)
   ((remove greasebands from trees) fruit4)
   ((thin peaches & nectarines) fruit4)
```

May

```
add((general-work5 veg fruit) May)

add((watch for greenfly) general-work5)
   ((kill slugs) general-work5)
   ((make successional sowings) general-work5)
   ((earth up early potatoes) general-work5)

add((plant out winter greens & celery) veg5)
   ((sow Dwarf French beans, beetroot) veg5)
   ((sow chicory, sweet corn, runner beans) veg5)

add((start to pick green gooseberries) fruit5)
   ((thin out raspberry canes) fruit5)
   ((cover strawberry beds) fruit5)
   ((plant melons in frames) fruit5)
```

June

```
add((general-work6 veg fruit) Jun)

add((earth up potatoes) general-work6)
   ((blanch leeks) general-work6)

add((thin out veg seedlings) veg6)
   ((plant outdoor tomatoes) veg6)

add((train and feed melons) fruit6)
   ((finish thinning peaches etc) fruit6)
```

July

```
add((general-work7 veg fruit) Jul)
```

```
      add((continue to thin seedlings) general-work7)
         ((cut Globe artichokes) general-work7)
         ((make successional sowings) general-work7)

      add((sow carrots, spring-cabbage, parsley) veg7)
         ((prune cherries & plums) fruit7)
         ((burn straw on strawberry beds) fruit7)
         ((start to pick apples and pears) fruit7)
```

August

```
      add((general-work8 veg fruit) Aug)

      add((control pests & diseases) general-work8)
         ((ripen off onions) veg8)
         ((make new strawberry beds) fruit8)
```

September

```
      add((general-work9 veg fruit) Sep)

      add((prevent mildew) general-work9)

      add((plant spring-cabbage) veg9)
         ((successional sowings) veg9)

      add((ripen melons in frames) fruit9)
         ((prune loganberries) fruit9)
```

October

```
      add((general-work10 veg fruit) Oct)

      add((gather fallen leaves) general-work10)
         ((protect cauliflowers) general-work10)

      add((plant spring-cabbage) veg10)
         ((prick off cauliflowers) veg10)

      add((collect fruit) fruit10)
```

November

```
      add((general-work11 veg fruit) Nov)

      add((dig all available ground) general-work11)
         ((successional sowings) general-work11)

      add((lift & blanch chicory) veg11)
         ((sow broadbeans) veg11)

      add((prune fruit trees) fruit11)
```

December

```
add((general-work12 veg fruit) Dec)

add((dig vacant ground) general-work12)

add((protect celery, cauliflower) veg12)

add((continue to prune fruit trees) fruit12)
```

Questions

To question January you can ask

```
which(x:x Jan)
```

and to check what general jobs should be done for this month you ask

```
which(x:x general-work1)
```

To ask about fruit and veg you type

```
which(y:y fruit1)
which(z:z veg1)
```

Other questions

```
which(x:x Dec)
which(x:x general-work12)
which(y:y fruit4)
```

You can also call up information about any kind of fruit in your garden

```
which(x: x fruit)
```

The data relating to this question is as follows.

Types of fruit

```
add((apples gooseberries pears plums
     strawberries raspberries) fruit)

add((Beauty of Bath (BB)) apple)
   ((Laxtons Epicure (LE)) apple)
   ((George Cave (GC)) apple)
   ((Discovery (D)) apple)
   ((Bramley (B)) apple)
   ((Edward VII (Ed)) apple)
```

```
add((small, crops late-mid summer, does not
     store) BB)
   ((medium, crops early autumn) D)
   ((large, crops late autumn, stores well, cooks
     well) B)

add((Cambridge Vigour (CV)) strawberry)
   ((Cambridge Favourite (CF)) strawberry)
   ((Redgauntlet (RG)) strawberry)
   ((Aromel (A)) strawberry)
   ((Royal Sovereign (RS)) strawberry)
   ((Talisman (T)) strawberry)
   ((Sonjana (SJ)) strawberry)

add((early crop; excellent for freezing, jam
     making) CV)
   ((perpetual crop; climber) SJ)
   ((late crop; vigorous plant, large fruit) T)
```

Questions

```
which(x:x apple)
which(x:x BB)
which(y:y strawberry)
which(Z:Z SJ)
```

You can also find out about types of vegetable by asking

```
which(x:x veg)
```

The data for this is as follows.

Types of vegetable

```
add((broadbean pea cabbage leek) veg)

add((Longpod) broadbean)
   ((Windsor) broadbean)
   ((Dwarf) broadbean)
   ((Kelvedon Wonder (KW)) pea)
   ((Little Marvel (LM)) pea)
   ((Lincoln (L)) pea)
   ((Gloriosa (G)) pea)
   ((Commander (C)) pea)

add((large, good crop: banned by EEC!) C)
   ((early crop: good for successional growing)
     KW)
```

Questions

```
which(x:x broadbean)
which(X:X pea)
which(y:y KW)
which(x:x C)
```

If you find something wrong with your peas you can ask

```
all(x:x symptom (maggots in pods))

Answer
   (Pea moth)
    No (more) answers
```

Pests and diseases

A guide to pea problems

```
add((Pea moth) symptom (maggots in pods))
   ((Pea weevil) symptom (holes around edges of
     leaves))
   ((Mildew) symptom (powdery white patches))
   ((Fungus) symptom (rotting stems))
```

Cures

```
add((Mildew) cure (spray with dinocap. Remove
     dead plants))
   ((Fungus) cure (dig up and burn: use good
     soil))
   ((Pea moth) cure (grow an earlier variety))
```

```
which(x:Fungus cure x)

Answer
    (dig up and burn: use good soil)
```

Before you begin gardening you should check the database to see what are the necessary conditions for a successful garden.

General information

```
add((Do you live in the north) indicates (Start
     your veg under glass or in polythene
     tunnels))
add((Do you live in the south) indicates (You can
     start planting/sowing in March))
```

```
      add((Do you have a source of manure) indicates
           (Best manure in order of quality is pigeon,
           chicken, goat, horse and cow.))
         ((Do you have a sheltered south-facing wall)
           indicates (A good place to try grapes,
           peaches and tomatoes))
         ((Do you have helpful animals) indicates
           (toads, hedgehogs and ducks eat slugs,
           snails, chrysalids etc.))
         ((Do you have hover-flies) indicates (These
           are useful insects that feed on aphids and
           other pests. You can attract them by
           planting buckwheat))
         ((Is your garden exposed) indicates (Wind
           prevents runner-beans, although you might be
           able to use a dwarf variety))

      x information if x indicates y and (x) is-told

Questions
      all(x y:x information and x indicates y)

Answer
      Do you live in the north? yes
      Do you live in the north?  Start your veg under
         glass or in polythene tunnels.
      Do you live in the south? no...
```

To use this database successfully you will need to make your own additions and modifications. A useful source of information can be obtained from the Soil Association or the Henry Doubleday Society. The addresses of these organizations may be found in most reference libraries.

Daily spread

Margarine was invented in France by Mege Mouries. There are basically two types, block and soft. There are others called low-fat spreads, but technically they cannot be called margarine. True margarine should consist of 80% fat and not more than 16% water.

All margarines contain fat. But not all fats are the same. For example you will find that fats are divided into

```
  ((animal coconut palm-oils) saturated-fatty-acids)
  ((herrings sardines some veg) polyunsaturated-acids)
  ((olive-oil and all fats) monounsaturated-acids)
```

For health reasons, some doctors say that you should avoid margarine or butter which contain saturated-fatty-acids.

Let´s compare some brand names by the amount of and type of fat that they contain.

Key to data
SF = saturated-fatty-acid (percentage)
PU = polyunsaturated-acid

```
Blocks        SF   PU Oil         Salt   Brand
       add((32   17 fish-veg   1.9)   Coop)
          ((35   24 fish-veg   2.0)   Kraft)
          ((44   11 beef-veg   2.6)   Krona)
          ((36   21 fish-veg   2.6)   Stork)

Softs     ((18   51 sunfl-veg  1.9)   Flora)
          ((22   54 sunfl-veg  1.7)   Safeway-Soft)
          ((20   21 fish-veg   2.1)   Stork)
          ((20   42 soya-veg   1.7)   Sainsbury-Soya)

Low-fat   ((20   33 veg        1.4)   Outline)
          ((15   21 veg        2.8)   Key-Slimmers)

Butter    ((74   3  butterfat  2.2)   Anchor)
```

```
      add((Coop Kraft Krona Stork) Block)
         ((Flora Safeway-Soft Sainsbury-Soya Stork)
           Soft)
         ((Outline Key-Slimmers) Low-fat)
         ((Anchor) Butter)
```

Questions
Which margarine has the lowest SF?
To answers this question we need to make a rule to define ´lowest-SF´ content.

```
x low-SF if (y|x1) x & (y1|y2) z & y LESS y1
```

Since SF is the first data entry in the list, we only require the head of the list y and y1.

```
which(x:x low-SF)
```

What does Flora contain?

```
all(x:x Flora)
```

How much salt in Anchor?

```
all(x:(x1 x2 x3 x) Anchor)
```

I have included one brand of butter for comparison.

Footballs

Do you want a football to kick around on the beach? Or are you a budding First Division Star? You could pay anything from £1 to well over £50 for a football. And there are many to choose from. This program sets out what you get for your money.

The footballs in the database were tested by a professional football club and a school football team. (Only two brands out of ten are given here.)

```
add((Professional club) tested-by)
   ((School team) tested-by)
```

```
which(x:x tested-by)
```

Professional and School team comments
Both teams commented on the balls they had tested.

```
((Pretty good ball: nice to train with; close to
   match standard) Adidas-Zephyr (Professional
   club))
((hard, no give; difficult to ´lift´)
   Adidas-Zephyr (School team))
((reasonable ball; bit ´puddingly´) Dunlop-Select
   (Professional club))
((heavy; hard; joint top-rated laminated ball)
   Dunlop-Select (School team))
```

The footballs were also tested under laboratory conditions for

1. punctures
2. abrasion
3. how well made
4. absorption of water
5. remaining inflated
6. compliance with FA regs (68 cm to 71 cm circumference: 396g to 453g weight; inflated pressure of 9 to 10.5 lb/in2)

Each ball was given a rating from 0 to 5.

Key to data

PU = punctures
SE = seams
AB = abrasion
AS = absorption

```
               PU   SE   AB   AS   Brand
         add((4    2    5    4  ) football  Adidas-Zephyr)
            ((4    5    4    4  ) football Dunlop-Select)
```

Rating the footballs

To find the total rating for each ball we need a rule.

```
x total-rating y if (x1 x2 x3 x4) football x &
   SUM(x1 x2 z) & SUM(z x3 z1) & SUM(z1 x4 y)
```

```
which(x y: x total-rating y)
```

Answer
?

But which football is the better choice?

```
x better-choice if x total-rating y and z
   total-rating Y and Y LESS y
```

```
which(x:x better-choice)
```

Answer
?

You and me - a social psychology program

Throughout this book you have been looking at logical relationships. You have seen a binary relationship such as ´Zorg likes Selina´ and a unary relationship such as ´Doc friend´.

Human society is also concerned with relationships between people. The essence of these relationships is communication.

Communication is what keeps people together. It is one of the most important parts of human behaviour. Whether you are a student of psychology or are just an observer of your fellow creatures, then the following program (written in Standard Syntax) might interest you.

There are many ways to communicate, for example there is non-verbal communication

```
((comms making-noises))
((comms the-face))
((comms the-body))
((comms clothes))
```

```
WHICH(X(comms X))
```

You will have to define WHICH yourself!

Personality and personal identity

```
((comms the-face))
```

Communication with one´s face can be quite sophisticated and meaningful.

Every face is unique. But is the data below a sign of personality?

```
((jolly fat-face))
((serious thin-face))
((conscientious thin-lips))
((aggressive thick-lips))
((intelligent high-forehead))
((excitable protruding-eyes))
((not-alert dull-eyes))
```

```
WHICH(X(intelligent X))
WHICH(X(excitable X))
WHICH(Y(jolly  Y))
```

Beware, appearances are deceptive!

Gestures also play an important part of communication.

```
((meaning head-nod agreement))
((meaning shake-fist anger))
((meaning rub-palms anticipation))
((meaning clapping approval))
((meaning yawn boredom))
((meaning rub-hands cold-or-delight))
((meaning fingers-crossed good-luck))
((meaning raise-hand attention))
((meaning shake-hands greeting))
((meaning pat-on-back encouragement))
((meaning flicking-the-chin bribery))
```

```
?((meaning yawn boredom))
WHICH(Z(meaning Z good-luck))
```

Gestures tell us much about a person. Some people talk with their hands. It is said that some Italians could not speak if they were handcuffed.

The use of space can tell us certain things about a person´s character and culture.

```
((family personal-space (18 inches)))
((friends social-space (4 - 9 feet)))
((religion Brahmins (7 feet)))
((religion Nayars (25 feet)))
```

```
WHICH(X(religion X (7 feet)))
?((family personal-space X))
WHICH(Y(friends X Y))
```

Certain castes in India must not come into contact with other castes for religious reasons. In the west our own family may come close to us, but friends are usually required to remain at a distance.

Touching is also another feature that can tell us about a person´s culture.

In each of the following cities, people sitting in a cafe were observed for one hour; the number of times people touched each other was recorded.

```
((San-Juan 180))
((Paris 110))
((Gainesville-USA 2))
((London 0))
```

```
?((London 10))
?((Paris 110))
WHICH(X(San-Juan X))
```

From the data provided so far, we can see that communication takes place not only by satellite and word of mouth, but occurs through a multitude of differing media.

Your personality

For this program you will need to record the following mannerisms

hands - excessive use
hands - little or no use
expanded posture (confident)
contracted posture (insecure)
space - close proximity to others
space - distant from others
touch - frequent in conversation
touch - infrequent if at all
voice - loud and hurried
voice - medium and unhurried

```
((behaviour Daniel (1 0 1 0 1 0 0 1 1 0)))
((behaviour Karen (0 1 0 1 0 1 0 1 0 1)))
((behaviour Steve (1 0 0 1 0 1 0 1 0 1)))
((behaviour Mike (1 0 1 0 1 0 1 0 1 0)))
((behaviour Zorg (0 1 1 0 1 0 0 1 0 1)))
```

To give meaning to the codes we need a rule.

```
((has x (excessive hand-use)) (behaviour x
   (1|y)))
((has x (little hand-use)) (behaviour x (y 1|z)))
((has x (expanded posture)) (behaviour x (y z
   1|X)))
((has x (contracted posture)) (behaviour x (y z X
   1|y1)))
((has x (close proximity)) (behaviour x (y z X y1
   1|z1)))
((has x (distant)) (behaviour x (y z X y1 z1
   1|X1)))
((has x (frequent touch)) (behaviour x (y z X y1
   z1 X1 1|y2)))
((has x (infrequent touch)) (behaviour x (y z X
   y1 z1 X1 y2 1|z2)))
((has x (loud & hurried)) (behaviour x (y z X y1
   z1 X1 y2 z2 1|X2)))
((has x (medium & unhurried)) (behaviour x (y z X
   y1 z1 X1 y2 z2 X2 1)))
```

Who has an expanded posture?

```
WHICH(x(has x (expanded posture)))
```

Whose voice is loud and hurried?

```
WHICH(y(has y (loud & hurried)))
```

Recording facts is a purely objective exercise. Drawing conclusions from your facts is quite another matter. This is where cultures, religions and different schools of social psychology disagree. Do not be misled by such statements as

He is a Mediterranean type
He is an extroverted Englishman
She has perfect composure

which are based on weak evidence. To show you what I mean, here are some rules

```
((Mediterranean-type x) (has x (1 0 1 0 1 0 1 0 1 0)))
((extrovert-English x) (has x (1 0 1 0 1 0 0 1 1 0)))
((has-perfect-composure x) (has x (0 1 1 0 1 0 0 1 0
   1)))
```

Questions

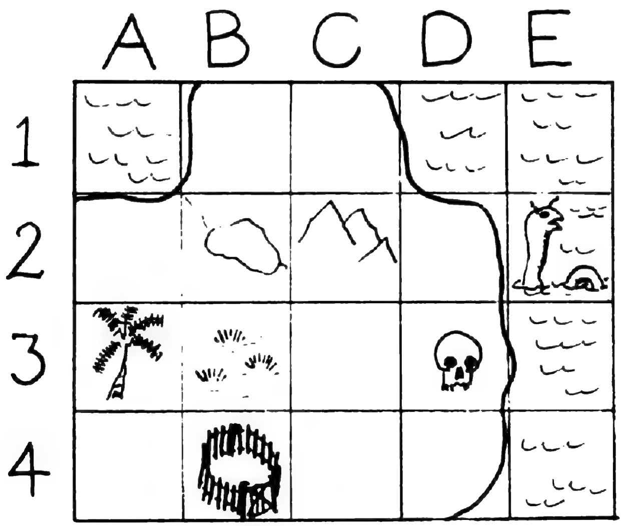
```
WHICH(X(Mediterranean-type X))
WHICH(x(extrovert-English x))
WHICH(Y(has-perfect-composure Y))
```

How would you describe yourself?

Treasure Hunt

You can use PROLOG to play games. One game that you can either play with a friend or with your computer is Treasure Hunt.

For this game you will need a piece of squared paper and a pencil. One player draws a treasure map on the square paper as illustrated.



The map is divided into 20 squares, each one having a grid reference. The other player has to draw the map by asking the computer questions about the map. When the map is complete, the player has to guess where the treasure is. For each correct answer for the map, the player scores 1 point. For each wrong answer the player scores -1 point. The player can only have three guesses to find the treasure.

Describing the map

Each square on the map is described in PROLOG as follows.

Type **accept in**

```
(sea (A1)) (sea (D1)) (sea (E1)) (sea (E2)) (sea
   (E4))

(land (B1)) (land (A2)) (land (A4))
(land (D2)) (land (D3)) (land (C2)) (land (C3))
   (land (C4))

(sea-serpent (E2))
(skull (D3))
(lake (B2))
(palm-tree (A3))
(swamp (B3))
(deserted-stockade (B4))
(mountains (C2)) end
```

Now type **list in**, and you should have the following data.

```
sea in A1
sea in D1
sea in E1...etc
```

The sentence to hide the treasure is

```
add(treasure found-at lake)
```

Questions

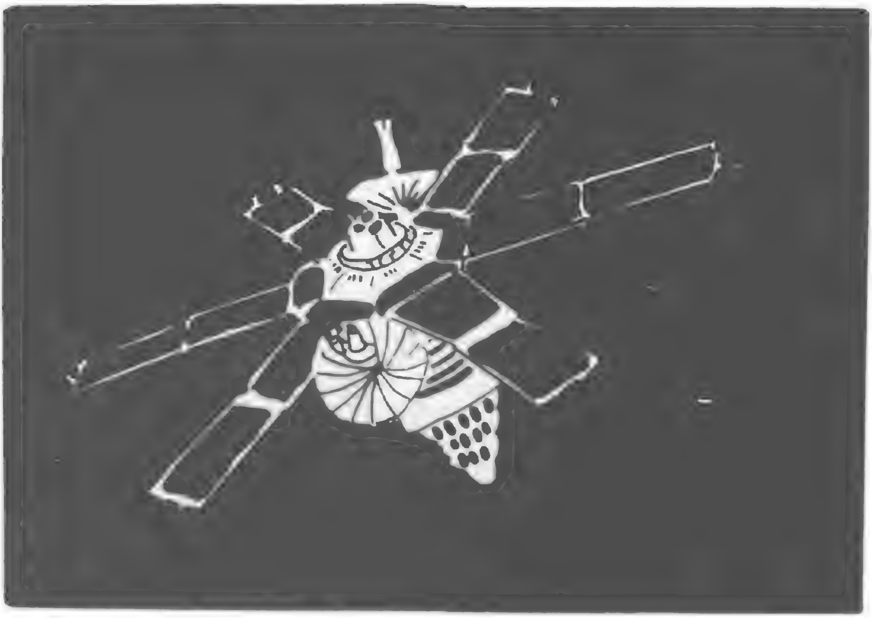
```
which(x:x in (A1))
```

Answer

```
land
```

```
is(treasure found-at skull)
NO
is(treasure found-at lake)
YES
```

Space probe



This program is a space simulation which involves exploring two new planets by means of a space probe. The probe can measure temperature, radiation and the presence of gases. To interpret the results obtained by the probe, you have to consider two important factors.

1. What do you understand by atmosphere?
2. What kind of atmosphere does the planet have, if any?

You could test to see if the planet has the following

1. twilight
2. a high level of radiation
3. extreme temperatures
4. clouds
5. gases (which ones are present?)

Below is data for the planets Solaris and Shaytan4. This data contains chemical and physical information about these planets. The information is compiled by means of APPEND.

Chemical and physical analysis

```
add(Solaris chem-analysis (O2 CO2 H2O N2))
   (Solaris phys-analysis ((0.02 radiation)
    (temp. 10-20C)))
```

```
(Shaytan4 chem-analysis (CO2 MH4 H2))
(Shaytan4 phys-analysis ((0.5 radiation)
 (temp. 300C)))
```

```
x report-on x1 if x1 chem-analysis y &
  x1 phys-analysis z & APPEND (z y x)
```

```
all(x:x report-on Solaris)
all(x:x report-on Shaytan4)
```

Knowing that the planets have certain gases may not be enough data for you to make a safe landing. You therefore need to ask the probe what is the percentage of each gas.

Percentage of gases

```
((ARR))
        add((Solaris O2 (20%))
           ((Solaris N2 (79%))
           ((Solaris CO2 (1%))

        add((Shaytan4 CO2 (70%))
           ((Shaytan4 MH4 (20%))
           ((Shaytan4 H2 (10%))
```

```
x1 safe-atmosphere if x1 O2 y & y LESS 50 &
  x1 CO2 z & z EQ 1
```

```
which(x:x safe-atmosphere)
is(Solaris safe-atmosphere)
is(Shaytan4 safe-atmosphere)
```

In the rule the safe level for CO2 is 1% and that for O2 is less than 50%. EQ is an in-built function for equals.

z EQ 1 means z equals 1

This simulation can of course be adapted to any situation. You could write an interactive simulation using is-told.

Appendix 4: PROLOG Commands

SIMPLE commands used in this book.

add
Allows you to add a sentence to your database.

```
add(z happy if z male and z not bachelor)
```

accept
Allows you to enter sentences without having to use add each time. It can be used with both unary and binary forms.

```
accept dangerous
(Chief-Nurk) (Nurk1) (Nurk2) (surface) end

accept likes
(Zorg Doc) (Nurk1 Nurk2) end
```

[This is equivalent to ´add(Zorg likes Doc)´ etc...]

delete
Used to delete sentences.

```
delete(Kolka eats garlic)
delete found-on 1
```

[This deletes the first ´found-on´ sentence.]

edit
(See Chapter 1, page 14)

EQ
Stands for equals.

is
Used to ask question requiring a YES/NO answer.

```
is(Minsk in Byelorussia)
is(Byelorussians live-in Manchester)
is(x lives-on surface)
```

isall
Takes sentences and puts them together in the form of a list.

```
            which(x:y isall(z:z friend-of Ted))
     Answer     (Sue Bob Cy Phil)
                No (more) answers
```

is-told
Sets up an interactive program, whereby the computer asks you questions. To use this command, LOAD TOLD on the cassette.

```
            Data
            bananas food
            grapes food

            which(x:x food (Do you like x) is-told)
     Answer     Do you like bananas  Yes
                Do you like grapes   No
```

kill
Deletes an entire relation or the whole program if you use kill all.

```
            kill likes
```

list
Displays all sentences with the same relation.

```
            list friend
```

list all
Displays the entire program on the screen.

load
Used to load your programs.

```
            load MYSTUFF
```

not
Negates sentences. It must be preceded by a positive sentence.

```
            x harmless if x big and not x dangerous
            x dangerous if x small and not(x likes z
              and x good-eye-sight)
```

one
Operates in a similar way to ′which′, but outputs only one answer at a time.

```
            one(x:x purple)
     Answer     Planet
                more? (y/n)
```

save
Used to save your programs.
```
          save MYSTUFF
```

SUM
Adds two numbers together. It can also be used to find the difference between two numbers.
```
          which(x: SUM(25 25 x))
    Answer    50

          which(y: SUM( 25 y 50))
    Answer    25
```

TIMES
Multiplies two numbers together. It can also be used to divide two numbers.
```
          which(x: TIMES(12 12 x))
    Answer    144

          which(z: TIMES(15 z 30))
    Answer    2
```

which
Attempts to find all the answers to your questions using variables. The forms of the questions are determined by the user.
```
          which(z:z likes Selina)
    Answer    Zorg
              Doc
              Capt-Tee
              Astra
              No (more) answers

          which(z y: z likes y)
    Answer    Zorg Selina
              Doc Selina
              Capt-Tee Selina
              Astra Selina
              No (more) answers

          which((x y): x lives-in y)
    Answer    (Sir-Michael Wimborne)
              No (more) answers
```

MICRO commands

(( ))
Brackets are not a command, but they are used to enter data sentences or clauses. Note that the relation word must be written first.

```
((is-in Varna Bulgaria))
```

which
Tries to find all the answers to your questions using variables. A colon (:) is not used as in SIMPLE.

```
which(x(is-in x Bulgaria))
```

is
Same as SIMPLE, but relation word written first.

```
is(is-in Varna Bulgaria)
```

Standard Syntax

(( ))
Double brackets are used to enter clauses as in MICRO.

```
((written-by Purple-Planet S.Gascoigne))
```

?
Used to question data. The answer, if true is indicated by the prompt sign &. If false another ? appears.

```
?((is-in Varna Bulgaria))
```

ADDCL
Adds a clause to the database. To be more specific ADDCL may be followed by a figure to indicate where the clause should be placed.

```
?(ADDCL((recipe Fish))1))
```

DELCL
Deletes clauses.

```
?((DELCL(recipe 1))
```

LIST
Lists clauses by relation word.

```
LIST recipe
```

LIST ALL
Lists entire program.

P
Prints text on the screen.

```
P(that´s all!)
```

PP
Prints text on the screen and starts a new line.

Index